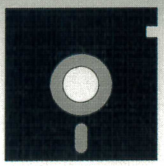
Markt&Technik

Said Baloui

Effektives Programmieren mit QuickBASIC

Eine systematische Anleitung zum Entwickeln von effizienten und professionellen Programmen unter Microsoft QuickBASIC.

Auf Diskette im PC-/MS-DOS-Format enthalten: alle Programm-Module aus dem Buch, wie z. B. Window-Verwaltung, Pull-down-Menüs, schnelle Bildschirmausgabe, Eingabemasken-Verwaltung und »CONVERT«, einem Programm, das Ihre GW-BASIC-Programme nach QuickBASIC konvertiert.



Effektives Programmieren mit QuickBASIC

Said Baloui

Effektives Programmieren mit QuickBASIC

Eine systematische Anleitung zum Entwickeln von effizienten und professionellen Programmen unter Microsoft QuickBASIC.

Markt & Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Baloui, Said:
Effektives Programmieren mit QuickBASIC : e. systemat. Anleitung
zum Entwickeln von effizienten u. professionellen Programmen unter Microsoft QuickBASIC / Said Baloui. –
Haar bei München : Markt-u.-Technik-Verl., 1987. – & 1 Diskette
ISBN 3-89090-532-3

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische
Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

QuickBASIC, MS-DOS und MS-BASIC sind eingetragene Warenzeichen der Microsoft Corp., USA

15 14 13 12 11 10 9 8 7 6 5 4 3 2
91 90 89 88

ISBN 3-89090-532-3

© 1988 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/West-Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Kösel, Kempten
Printed in Germany

Inhaltsverzeichnis

Vorwort 9

Einleitung 13

1 Grundlegende Eigenschaften 17
1.1 Label statt Zeilennummern 17
1.2 Kompilieroptionen 19
1.3 Ausgabeoptionen 21
1.4 Mehrzeilige Anweisungen 21
1.5 Erweiterte IF..THEN..ELSE-Struktur 23
1.5.1 IF..THEN..ELSE-Blockstruktur 23
1.5.2 ELSEIF: Auswahl unter alternativen Bedingungen 24
1.6 Benutzerdefinierte Funktionen 27
1.6.1 Praktische Anwendung von Funktionen 30

2 Prozeduren 33
2.1 Aufbau und Aufruf einer Prozedur 33
2.2 Interpreter- und Compilersprachen 35
2.3 Variablenorganisation 36
2.3.1 Lesezugriff 37
2.3.2 Schreibzugriff 37
2.4 Lokale Variablen 38
2.5 Übergabe von Variablen 42
2.5.1 Wertübergabe 43
2.5.2 Adreßübergabe (Übergabe nach Referenz) 46
2.5.3 Übergabe komplexer Ausdrücke 49
2.5.4 Übergabe von Arrays 51
2.6 Globale Variablen 55
2.6.1 Die Anweisung COMMON 56
2.6.2 Das Attribut SHARED 57
2.6.3 Die Anweisung STATIC 59

3	**Entwicklung einer Routinen-Sammlung**	63
3.1	Der Metabefehl $INCLUDE	63
3.2	Deklaration globaler Variablen (COMDEF.BAS)	65
3.3	Initialisierung globaler Variablen (INIT.BAS)	68
3.4	Die Module	69
3.5	Listing von LIBRARY.BAS	80
4	**User-Libraries**	83
4.1	Erstellung einer User-Library	85
4.2	Ein Standardfehler	90
4.3	Globale Variablen in User-Libraries	92
4.4	Einbindung vorhandener Module	95
4.5	Das getrennte Kompilierverfahren	99
4.5.1	Systemeinrichtung	101
5	**Eingaberoutine**	107
5.1	Programmentwicklung	111
5.2	Listing der Eingaberoutine (INPUT.BAS)	124
5.3	Einbindung in die User-Library	127
5.4	Aufruf der Eingaberoutine (INPUTDEM.BAS)	128
6	**Maskensteuerung**	133
6.1	Programmentwicklung	135
6.2	Listing der Maskensteuerung (MASKE.BAS)	140
6.3	Einbindung in die User-Library	141
6.4	Aufruf der Maskensteuerung (MASKDEMO.BAS)	141
7	**Windowing und Pull-down-Menüs**	143
7.1	Schnelle Bildschirmausgabe mit PRINTF	144
7.2	Programmstruktur	147
7.3	Definition der Menüs	149
7.4	Verwaltungsmodule	152
7.5	Programmentwicklung	153
7.6	Statische und dynamische Arrays	181
7.7	Listing der Pull-down-Menüs (PULLDOWN.BAS)	189
7.8	Einbindung in die User-Library	195
7.9	Aufruf der Pull-down-Menüs (PULLDEMO.BAS)	197
7.10	Aufruf der Window-Routinen	199
7.11	Listing von WINDOWS.BAS	202

8	**Konvertieren von GW-BASIC-Programmen**	205
8.1	Features von CONVERT	207
8.2	Grundlegende Routinen	209
8.3	Verwendete Datenstrukturen	218
8.4	Programmentwicklung	221
8.4.1	Globale Variablen in komplexen Programmsystemen	221
8.4.2	Fehlerbehandlung	224
8.4.3	Redimensionieren dynamischer Arrays	228
8.5	Listing des Konvertierungsprogramms (CONVERT.BAS)	253
8.6	Anwendung von CONVERT	263
9	**Lauffähige Programme mit LINK.EXE erzeugen**	265
9.1	Voraussetzungen für den Ablauf eines Programms	265
9.2	Linken mit BRUN20.LIB	268
9.3	Linken mit BCOM20.LIB	272
9.4	Die optimale Systemeinrichtung	277
10	**Strukturierte Programmierung mit der Version 3.0**	279
10.1	Konstanten	279
10.2	Schleifenstrukturen	280
10.3	Vorzeitiges Verlassen von Schleifen	282
10.4	Auswahl unter alternativen Bedingungen	285
11	**QuickBASIC und Assembler**	289
11.1	Assembler-Programme einbinden	289
11.2	Stackverwaltung	293
11.3	Standardrahmen für Assembler-Routinen	295
11.4	Parameterübergabe	296
11.5	Schnelle Bildschirmausgabe: PRINTF	302
11.6	Listing von PRINTF.ASM	308
11.7	Windowing: WINDOWS	310
11.8	Listing von WINDOWS.ASM	315
Anhang		321
Programme auf der beiliegenden Diskette		321
Stichwortverzeichnis		324
Hinweise auf weitere Markt&Technik-Produkte		329

Für Maria, meinen privaten Lektor und Tester.
Danke für Deine Geduld mit einem Programmiersüchtigen.

Vorwort

Wenn Sie sich für dieses Buch interessieren, haben Sie wahrscheinlich noch vor kurzem in GW-BASIC programmiert und besitzen QuickBASIC noch nicht allzu lange. Für kleinere Programme ist GW-BASIC völlig ausreichend, Sie wollen anscheinend dennoch »mehr«.

Sie gehören also zu jenen Programmierern, die größere Programme erstellen und entsprechend hohe Ansprüche an Ablaufgeschwindigkeit und Strukturierung ihrer Programme stellen.

Vor QuickBASIC konnten diese Ansprüche nur »strukturierte« Sprachen wie Pascal oder Modula erfüllen. Nun, QuickBASIC ist ohne Zweifel ebenfalls eine strukturierte Sprache und nicht mit den bisherigen einfachen BASIC-Versionen zu vergleichen.

Ich gehe davon aus, daß Sie einen »einfachen« Dialekt (GW-BASIC ?) gut kennen und werde Ihnen daher auch nicht zum tausendsten Mal PRINT und INPUT erläutern. Stattdessen zeige ich Ihnen, wie Sie mit QuickBASIC möglichst effektiv größere Programmsysteme entwickeln können.

Sollten Sie gerade Ihre ersten Versuche mit BASIC hinter sich haben, rate ich Ihnen von diesem Buch ab. Es setzt fundierte BASIC-Kenntnisse voraus. Erläuterungen der Standardbefehle werden Sie nicht finden – die bieten die Handbücher zu QuickBASIC und GW-BASIC.

Da ich eine Abneigung gegen theoretische Lehrbücher habe, werden alle Erläuterungen von Programmbeispielen begleitet. Die ersten Kapitel behandeln die grundlegenden Eigenschaften von QuickBASIC. Nach Erarbeitung der Grundlagen entwickeln wir umfangreichere und komplexere Programme, die Ihnen zeigen, wie die speziellen Fähigkeiten von QuickBASIC möglichst effektiv eingesetzt werden.

Diese komplexeren Programme demonstrieren nicht nur den Umgang mit QuickBASIC. Sie sind zugleich vollwertige Routinen, die in jedem(!) professionellen Programm benötigt werden. Alle Programme befinden sich auf der beiliegenden Diskette und können von Ihnen problemlos in Ihren eigenen Programmen benutzt werden.

Zum Aufbau: Im ersten Teil erläutere ich die »oberflächlichen« Unterschiede zwischen QuickBASIC und herkömmlichen BASIC-Versionen: die Programmerstellung ohne Zeilennummern, den Einsatz von Label und die nun sehr vielseitige IF-Struktur.

QuickBASIC wirklich auszunutzen, erfordert jedoch eine enorme Umstellung gegenüber GW-BASIC. Ohne fundiertes Wissen darüber, wie ein Compiler arbeitet, wie QuickBASIC den Speicher verwaltet, was mit »unabhängigem Objektmodul«, unter »Variablenübergabe nach Wert/Referenz« oder dem »Linken von Modulen« gemeint ist, sind Schwierigkeiten bei größeren Programmprojekten unvermeidbar.

Der zweite Teil des Buches beschäftigt sich mit diesen Eigenschaften von QuickBASIC, die für Compiler typisch sind. Wenn Sie nur den Umgang mit BASIC-Interpretern gewohnt sind, führen diese Eigenschaften zu einer Vielzahl von Problemen, auf die das Handbuch kaum eingeht. In diesem Teil werden zwei wichtige Eigenschaften von QuickBASIC intensiv behandelt: die Einbindung von INCLUDE-Files und vor allem der Umgang mit globalen und lokalen Variablen.

Der Hauptteil des Buches behandelt eine völlig neue Eigenschaft von QuickBASIC: die Erstellung sogenannter »User-Libraries«. Auf diese User-Libraries wird im Handbuch kaum eingegangen. Dabei ist gerade dieses »Feature« für die Entwicklung großer Programmsysteme außerordentlich nützlich und – bisher – bei keinem anderen Compiler verwirklicht.

Die Erstellung von User-Libraries bietet dem »Compiler-Neuling« eine Vielzahl möglicher Fehlerquellen. Im dritten Teil werden wir gemeinsam eine recht umfangreiche User-Library erstellen. In diese User-Library werden alle bis zu diesem Zeitpunkt entwickelten Routinen eingebunden. Diese Library oder auch »Bibliothek« werden wir anschließend zur Entwicklung eines größeren Programmprojekts verwenden. Nach der endgültigen Fertigstellung erläutere ich an diesem Programm die verschiedenen Kompilieroptionen und vor allem den Umgang mit dem »Linkprogramm« LINK.EXE.

Der letzte Teil wendet sich an die Assembler-Programmierer. Er erläutert die Einbindung von Assembler-Routinen. Zwei sehr nützliche Assembler-Routinen werden entwickelt, die zeigen, welche Probleme es bei der Parameterübergabe von QuickBASIC an Assembler gibt und wie sie gelöst werden.

Zusammengefaßt: Wiederholungen des Handbuches werden Sie in diesem Buch nicht entdecken. Den Umgang mit dem Editor, die Such- und Ersetzfunktionen werden in den Originalunterlagen sehr ausführlich beschrieben.

Statt dessen erwartet Sie eine problemorientierte Darstellung von QuickBASIC, wobei allgemeine BASIC-Grundlagen vorausgesetzt werden. Problemorientiert heißt: Sie lernen durch die Entwicklung immer komplexer werdender Programme, welche Möglichkeiten – und auch welche Probleme(!) – QuickBASIC bietet. Die entwickelten Programme sind alles andere als »Spielereien«. Sie werden in jedem professionellen Programm ständig benötigt und stehen Ihnen auf der beiliegenden Diskette für eigene Entwicklungen zur Verfügung.

Said Baloui

Einleitung

Im Gegensatz zu reinen Befehlserläuterungen ist dieses Buch mehr eine Art »Kursus«. Die einzelnen Kapitel bauen aufeinander auf. Daher ist es nicht empfehlenswert, einfach darin »herumzublättern« und größere Teile zu übergehen.

Eine Ausnahme bildet der Abschnitt »Grundlegende Eigenschaften von QuickBASIC«. Diesen Teil dürfen Sie getrost übergehen, wenn Sie sich bereits mit Label, mit den verschiedenen Kompilieroptionen und den sehr vielseitigen IF-Strukturen von QuickBASIC auskennen.

Sie sollten dieses Buch nicht nur darum komplett lesen, weil die einzelnen Kapitel aufeinander aufbauen. Es gibt noch einen anderen Grund. Das Buch ist nicht streng aufgeteilt in einen Theorie- und einen Praxisteil.

Es ist keineswegs so, daß zuerst alle neuen Eigenschaften von QuickBASIC theoretisch erläutert und dann nur bereits bekannte Theorie praktisch umgesetzt wird. In den praktischen Teilen wird nicht nur ein Programm entwikkelt, sondern immer wieder ist ein wenig Theorie eingestreut, in der neue, nützliche Eigenschaften von QuickBASIC erläutert werden. Jedes Programmprojekt ist also eine Mischung aus Theorie und Praxis.

Gute Beispiele dafür sind die Kapitel »Statische und dynamische Arrays« und »Redimensionierung dynamischer Arrays«. Beide Kapitel sind Teil größerer Abschnitte, in denen jeweils ein Projekt im Mittelpunkt steht. Und diese Programme verwenden nicht nur bis dahin bereits bekannte, sondern auch neue Eigenschaften von QuickBASIC. Das zu entwickelnde Programm ist dann der »Aufhänger« zur Erläuterung dieser neuen »Features«.

Programme auf der beiliegenden Diskette

Alle größeren Programme, die wir entwickeln – und auch kleinere Demoprogramme – befinden sich auf der Diskette zum Buch. Um Ihnen einen besseren Überblick zu verschaffen, sind die Programme in Gruppen aufgeteilt. Jede Gruppe befindet sich in einem eigenen Verzeichnis:

- DEMOS: Kleinere Demoprogramme
- ASEMBLER: Assembler-Programme
- BATCHES: Stapeldateien
- MODULE: Unsere »Großprojekte«
- OBJFILE: Objektdateien

Die Diskette enthält eine ganze Menge an Programmen. Sollten Sie trotz der Unterverzeichnisse einmal die Übersicht verlieren, schauen Sie einfach im Anhang dieses Buches nach. Sie finden dort eine Liste aller auf der Diskette vorhandenen Programme mit einer Kurzbeschreibung der Funktion.

Schreibkonventionen

Die Syntax von Befehlen wird auf die gleiche Weise wie im Handbuch dargestellt. Einige Besonderheiten gibt es jedoch, wenn ich mich im Text auf Variablen-, Prozedur- oder Dateinamen beziehe.

- Variablennamen befinden sich in eckigen Klammern:
  - <Nr>
  - <EditFlag>
- Prozedurnamen, Dateinamen und einzelne Anweisungen sind großgeschrieben:
  - LINPUT
  - MASKDEMO.BAS
  - $INCLUDE
- Auszüge aus Programmlistings werden in Listingschrift hervorgehoben:
  - `If PosAkt > 1`
  - `While Instr(Back$, Last$) = Ø`

Abweichende Begriffsverwendung

Im Bemühen, alles, aber auch wirklich alles »einzudeutschen«, werden im Handbuch zum Teil schauderhafte Begriffe verwendet. Was halten Sie von »Datenfeldvariablen« statt den gewohnten »Arrayvariablen« und von »Zeichenfolgevariable« statt »Stringvariable«? Wahrscheinlich ebensowenig wie ich. Programmierer sind nun mal an englische Bezeichnungen gewohnt und daran werde ich mich im Buch halten. Die wichtigsten Unterschiede zum QuickBASIC-Handbuch:

Dieses Buch	QuickBASIC-Handbuch:
Array	Datenfeld
String	Zeichenfolge
Library	Bibliothek
User-Libraries	Benutzer-Bibliotheken
Prozedur	Unterprogramm
Unterprogramm	Unterroutine

Etwas verwirrend im QuickBASIC-Handbuch ist die Verwendung der Begriffe »Unterprogramme« und »Unterroutinen«. »Unterroutinen« bezeichnen jene Programmteile, die Sie als »Unterprogramme« kennen, die mit GOSUB aufgerufen werden und mit RETURN enden. Da ich mich an dieser Begriffsverwirrung nicht beteiligen will, heißen Unterprogramme in diesem Buch auch weiterhin Unterprogramme und nicht »Unterroutinen«.

Um die Verwirrung komplett zu machen, wird im Handbuch der Ausdruck »Unterprogramm« auf einmal für eine völlig neue Struktur verwendet, die QuickBASIC zur Verfügung stellt und die in verschiedenen Programmiersprachen unter dem Begriff »Prozedur« bekannt ist. Auch ich werde für diese Programmteile den Ausdruck »Prozedur« verwenden.

Unterschiedliche QuickBASIC-Versionen

QuickBASIC gibt es schon relativ lang. Und wie üblich bei Programmen, die es schon länger gibt, existieren auch verschiedene Versionen, zum Beispiel die Versionen 1.0, 2.0, 2.01 und neuerdings 3.0.

Verschiedene Versionen von Anwenderprogrammen wie einer Textverarbeitung sind »unkritisch«. Neuere Versionen bieten entsprechend mehr Funktionen und sind fast immer schneller als die alte Version.

Bei einer Programmiersprache wie QuickBASIC fragen Sie sich bestimmt, ob Sie mit den Programmen in diesem Buch überhaupt arbeiten können, ob sie mit Ihrer Version zusammenarbeiten!

Keine Sorge, die Antwort lautet Ja, und zwar ohne jede Einschränkungen! Die Programme in diesem Buch wurden mit der Version 2.01 erstellt. Diese Version ist vollkompatibel mit den Vorgängerversionen, der Befehlsumfang identisch. Der Hauptunterschied: 2.01 ist die erste deutsche Version mit deutschen Menüs und Fehlermeldungen. Und dieser Unterschied soll Sie nun wirklich nicht weiter interessieren. Für Sie ist nur wichtig, daß Ihre Version über alle von mir verwendeten Anweisungen verfügt – und das ist mit Sicherheit der Fall.

Sollten Sie über den Nachfolger meiner eigenen Version von QuickBASIC verfügen, über die Version 3.0, lesen Sie bitte das Kapitel »Neue Features der Version 3.0«. Dieses Kapitel zeigt anhand einiger Programme, die wir bis dahin entwickelt haben, wie Sie die zusätzlichen Anweisungen in der Praxis einsetzen.

Noch ein wichtiger Hinweis für Besitzer der Version 3.0. Außer QuickBASIC selbst (QB.EXE) werden wir folgende »Systemdateien« verwenden: BRUN20.LIB, BRUN20.EXE und BCOM20.LIB. Bei der Version 3.0 ist die Zahl 20 im Dateinamen durch 30 ersetzt worden. Verwenden Sie also bitte statt dessen die Dateien BRUN30.LIB, BRUN30.EXE und BCOM30.LIB.

1

Grundlegende Eigenschaften

Der folgende Abschnitt stellt einige grundlegende Eigenschaften von QuickBASIC vor. Wenn Sie bereits im Umgang mit QuickBASIC geübt, mit der Programmerstellung ohne Zeilennummern und mit den Kompilieroptionen vertraut sind, sollten Sie dennoch spätestens bei der Behandlung der erweiterten IF-Anweisung wieder einsteigen.

Alle in diesem Abschnitt vorgestellten Demoprogramme finden Sie auf der Diskette zum Buch im Verzeichnis DEMOS.

1.1 Label statt Zeilennummern

BASIC ohne Zeilennummern ist auf den ersten Blick kaum vorstellbar. Dank der Zeilennummern konnten Sie in GW-BASIC wie im folgenden Beispiel problemlos nachträglich Anweisungen in ein Programm einfügen.

```
100 For i=1 to 100
110 Next i
105   Print i;
```

Sie alle wissen, daß BASIC-Interpreter Zeilen automatisch in aufsteigender Ordnung sortieren, so daß sich nach LIST folgendes Bild ergibt:

```
100 For i=1 to 100
105   Print i;
110 Next i
```

In QuickBASIC sind Zeilennummern »optional«, das heißt zulässig, aber nicht notwendig. Das kleine Beispielprogramm ohne Zeilennummern:

```
For i=1 to 100
  Print i;
Next i
```

Dank des komfortablen Editors von QuickBASIC ist das nachträgliche Einfügen von Anweisungen kein Problem. Entweder bewegen Sie den Cursor

auf den Beginn der nächsten Zeile und drücken RETURN, oder aber Sie verwenden die Tastenkombination CTRL+N, um eine Leerzeile einzufügen.

Nähere Erläuterungen zum Editor erspare ich Ihnen, da dieser in den Originalunterlagen zu QuickBASIC ausführlich beschrieben wird.

Nun fragen Sie sich vielleicht, wie ohne Zeilennummern »Sprungziele« angegeben werden. Den Anweisungen GOTO und GOSUB folgt üblicherweise eine Zeilennummer, die das Sprungziel definiert.

Zu diesem Zweck stellt Ihnen QuickBASIC »Label« zur Verfügung, im Handbuch »Zeilenmarke« genannt. Ein Label ist ein alphanumerisches Wort, das am Anfang einer Zeile steht und dem ein Doppelpunkt folgt, zum Beispiel:

```
SORT:
```

Statt einer Zeilennummer können Sie als Sprungziel ein Label angeben.

```
For i=1 to 100     REM Ein Kommentar mit »REM«
  gosub Ausgabe    '   Ein Kommentar mit »'«
Next i
End

ausgabe:
Print "Hallo";
Return
```

Dieses kleine Programm befindet sich im Verzeichnis DEMOS unter dem Namen LABEL.BAS. Es demonstriert einige Eigenschaften von QuickBASIC, mit denen Programme »optisch strukturiert« werden, die also die Lesbarkeit erhöhen:

- Verschiedene Programmteile können Sie durch beliebig viele Leerzeilen voneinander trennen.
- Kommentare dürfen wahlweise wie bisher mit REM oder aber mit einem einfachen Anführungszeichen beginnen. Kommentare können der letzten Anweisung einer Zeile unmittelbar folgen, ohne durch einen Doppelpunkt von dieser getrennt zu werden.
- Ob Sie für Anweisungen, Variablennamen und Label Groß- oder Kleinschreibung verwenden, ist für QuickBASIC – ebenso wie für GW-BASIC – uninteressant.

Dieses Buch soll Ihnen zeigen, wie mit QuickBASIC professionell programmiert wird. Aus diesem Grund werden Sie in den entwickelten Programmen weder Zeilennummern noch GOTO- oder GOSUB-Anweisungen finden.

Zeilennummern sind (wie erläutert) bei QuickBASIC absolut unnötig und verschlechtern nur die Lesbarkeit von Programmen.

Die Anweisungen GOTO und GOSUB sind hauptverantwortlich für den Ruf von BASIC, nur eine »Einsteigersprache« zur Erzeugung von unverständlichem »Spaghetti-Code« zu sein. Sie werden sehen: Mit neuen Sprachelementen wie IF..ELSEIF..END IF und den »Prozeduren« sind beide Anweisungen überflüssig geworden.

1.2 Kompilieroptionen

Das kleine Demoprogramm aus dem letzten Abschnitt sollten Sie versuchsweise eingeben. Da QuickBASIC ein Compiler ist, ist der »Quelltext« nicht unmittelbar ausführbar. Er muß kompiliert werden.

Entweder drücken Sie die Funktionstaste F5 (=Programm kompilieren) und anschließend die Tastenkombination CTRL+R (=Programm ausführen), oder Sie wählen die gleichen Funktionen über die Pull-down-Menüs an. Übrigens: CTRL+R bedeutet nicht nur »Ausführen«. QuickBASIC prüft selbständig, ob der Quelltext seit der letzten Kompilation geändert wurde. Wenn ja, wird auf das Kommando CTRL+R hin der geänderte Text erneut kompiliert und anschließend ausgeführt. CTRL+R ist somit das Kommando zum »Text – wenn nötig – kompilieren und anschließend ausführen«.

Nach der Kompilation befindet sich im Hauptspeicher Ihres Rechners außer dem Quelltext ein daraus erzeugtes Maschinenspracheprogramm. Dieses Programm liegt in unmittelbar für den Prozessor verständlicher Form vor. Es muß nicht mehr Anweisung für Anweisung interpretiert werden. Resultat: Die Ablaufgeschwindigkeit ist um ein Vielfaches höher als die des interpretierten BASIC-Programms.

Nach jeder noch so winzigen Programmänderung muß das komplette Programm neu kompiliert werden. Da QuickBASIC jedoch sehr schnell kompiliert, ist dieser Nachteil erst bei sehr umfangreichen Programmen erkennbar. Und genau dann kommen die erwähnten User-Libraries ins Spiel, mit denen wir uns noch intensiv beschäftigen werden. Sie beschleunigen häufige Kompiliervorgänge erheblich, da nur der geänderte Programmteil kompiliert werden muß.

Auf die Art und Weise, wie kompiliert wird, haben Sie großen Einfluß. Das Kompiliermenü – Anwahl über das Pull-down-Menü »Betrieb« und das darin enthaltene Kommando »Kompilieren...« stellt Ihnen eine Vielzahl von Optionen zur Verfügung.

Beim Kompilieren mit F5 beziehungsweise mit CTRL+R verwendet QuickBASIC normalerweise die Standardoptionen »Austesten«, »Zeichenfolgedaten minimieren«, »Kompilieren im Speicher«, und »Geschwindigkeit«. Im Kompiliermenü können Sie beliebige andere Optionen einstellen.

Alle Optionen sind in den Unterlagen zu QuickBASIC ausführlich beschrieben. Die Kompilierungsoptionen »Austesten«, »On Error«, »Resume Next«, »Ereigniserfassung« und »Prüfen zwischen Anweisungen« besitzen eine Gemeinsamkeit: Ist eine – oder mehrere – dieser Optionen aktiviert, werden langsamere und teilweise auch umfangreichere Programme erzeugt als bei Inaktivierung der Kompilierungsoptionen.

Bei einigen Optionen besitzen Sie keine Wahl. Wenn Ihr Programm ON ERROR oder RESUME NEXT verwendet, muß die entsprechende Option aktiviert sein. Wartet Ihr Programm auf das Auftreten von Ereignissen, die mit ON (Ereignis) GOTO (Zeilennummer/Label) abgefragt werden (zum Beispiel ON TIMER GOTO 1000), muß eine der beiden Optionen »Ereigniserfassung« oder »Prüfen zwischen Anweisungen« aktiv sein. Wenn diese Optionen nicht benötigt werden, sollten sie zugunsten schnellerer Programme inaktiviert sein.

Anders bei »Austesten«. Diese Option besitzt mehrere Funktionen:

1. Ohne »Austesten« kann sich der Rechner »aufhängen«, wenn während eines Programmlaufs Fehler auftreten wie zum Beispiel die Überschreitung von Arraygrenzen (<A$(..)> wurde mit DIM A$(100) dimensioniert, während des Programmlaufs werden jedoch höhere Indizes verwendet). Wenn Sie Pech haben, hilft dann nur noch die Betätigung des berühmten roten Schalters am Rechner. Mit aktiviertem »Austesten« erhalten Sie statt dessen eine »Laufzeit-Fehlermeldung«, zum Beispiel »Index außerhalb des Bereichs«, landen anschließend im Editor und können den Fehler sofort beheben.

2. »Austesten« erlaubt jederzeit den Abbruch eines Programms mit CTRL+BREAK.

3. Die Programmverfolgung mit TRON und TROFF ist nur möglich, wenn das Programm mit »Austesten« kompiliert wurde.

Da auch »Austesten« langsamere und umfangreichere Programme erzeugt, sollten Sie so vorgehen:

- Wenn sich ein Programm noch in der Entwicklungs- und Testphase befindet, kompilieren Sie es mit »Austesten«.

- Steht einwandfrei (!) fest, daß das Programm fehlerfrei ist, kompilieren Sie es ein letztes Mal ohne »Austesten«.

1.3 Ausgabeoptionen

Die Voreinstellung ist »Speicher«. Das Programm wird ohne Diskettenzugriffe kompiliert, das »Kompilat« – das erzeugte Maschinenspracheprogramm – befindet sich danach ebenfalls im Speicher. Während der Entwicklung eines Programms ist »Speicher« die zweckmäßigste Option.

Ein Nachteil ist jedoch, daß das Kompilat nicht einfach gespeichert werden kann und beim Ausschalten des Rechners verloren geht. Wenn Sie das Programm am nächsten Tag noch mal ausprobieren wollen, müssen Sie die gesamte Prozedur wiederholen: Quelltext laden, kompilieren, ausführen.

Die Option »EXE« speichert das Kompilat als .EXE-Datei auf der Diskette. Diese Datei wird nach dem Aufruf vom Betriebssystem aus sofort ausgeführt. Voraussetzung: das »Laufzeitmodul« BRUN20.LIB muß sich ebenfalls auf der Diskette befinden. BRUN20.LIB enthält alle Unterprogramme, die zur Ausführung der in Ihrem Programm enthaltenen BASIC-Anweisungen benötigt werden; ein Unterprogramm zur Ausführung der PRINT-Anweisung, ein Unterprogramm, das für INPUT zuständig ist und so weiter.

»Obj (BCOM.LIB)« und »Obj (BRUN.LIB)« sind die interessantesten, aber auch kompliziertesten Ausgabeoptionen.

Beide Optionen werden wir später besprechen. Bei den folgenden Programmentwicklungen gehe ich davon aus, daß Sie die Standardoption »Speicher« verwenden.

1.4 Mehrzeilige Anweisungen

GW-BASIC begrenzt die Länge einer Programmzeile auf maximal 255 Zeichen. Mehrere Anweisungen in einer Programmzeile müssen durch Doppelpunkte getrennt werden. Eine Bedingung mit einer umfangreichen Folge von Anweisungen im THEN- oder ELSE-Zweig paßt möglicherweise nicht mehr in eine Programmzeile und erfordert sehr umständliche Konstruktionen. Aber auch kürzere IF..THEN..ELSE-Anweisungen besitzen ihre Tücken:

```
100 If a=1 Then For i=1 to 10: Print "a besitzt den Wert 1": Next i Else
For i=1 to 10: Print "a besitzt einen Wert ungleich 1": Next i
```

Dieses »Programm« entspricht sicher nicht dem Ideal eines gut strukturierten und lesbaren Programms. Mit herkömmlichen BASIC-Dialekten sind derart unverständliche Programme leider manchmal nicht zu vermeiden. In QuickBASIC kann eine Programmzeile »mehrzeilig« sein. Eine »logische«

Programmzeile wird auf mehrere »Bildschirmzeilen« aufgeteilt. Voraussetzung: Jede Bildschirmzeile muß mit einem »Unterstrich« enden ("_")!

Struktur mehrzeiliger Bedingungen:

```
If Bedingung THEN_
  Anweisung:_
  ...
  Anweisung_
Else_
  Anweisung:_
  ...
  Anweisung
```

Angewandt auf unser Beispiel ergibt sich folgendes Programm, das Sie unter dem Namen LONG_IF.BAS im Verzeichnis DEMOS finden:

```
If a=1 Then_
  For i=1 to 10:_
    Print "a besitzt den Wert 1":_
  Next i_
Else_
  For i=1 to 10:_
    Print "a besitzt einen Wert ungleich 1":_
  Next i
```

»Mehrzeiler« dürfen beliebig lang sein. Mit dieser Methode können – wie im Beispielprogramm – auch Schleifen strukturiert werden, die sich innerhalb eines IF..THEN..ELSE-Statements befinden.

Vergessen Sie auf keinen Fall den Doppelpunkt zur Trennung mehrerer Anweisungen in einer »logischen« Programmzeile! Ob sich dieser trennende Doppelpunkt vor dem Unterstrich befindet oder aber am Anfang der nächsten Bildschirmzeile, ist unwichtig:

```
If a=1 Then_
  For i=1 to 10_
    :Print "a besitzt den Wert 1"_
  :Next i_
Else_
  For i=1 to 10_
    :Print "a besitzt einen Wert ungleich 1"_
  :Next i
```

Den Doppelpunkt erwähne ich, weil Mehrzeiler dazu »verführen«, ihn zu vergessen. Im abgebildeten Programm sind die Anweisungen scheinbar (!) voneinander getrennt, da sie sich in verschiedenen Bildschirmzeilen befinden. Diese optische Trennung der Anweisungen wird von QuickBASIC nicht nachvollzogen. Denken Sie daran: der gesamte Block ist für QuickBASIC eine (!) zusammenhängende »logische Zeile«. Verschiedene Anweisungen in einem Mehrzeiler sind auch weiterhin durch Doppelpunkte zu trennen.

Das nächste Demoprogramm zeigt noch einmal, wie QuickBASIC auf den Unterstrich »reagiert«:

```
Pr_
int "Guten Tag"
```

Sobald QuickBASIC auf den Unterstrich trifft, erwartet der Compiler die Fortsetzung der Anweisung in der nächsten Bildschirmzeile.

1.5 Erweiterte IF..THEN..ELSE-Struktur

QuickBASIC bietet einige Erweiterungen der IF..THEN..ELSE-Anweisung, »Blockstrukturen« und die Auswahl unter mehreren alternativen Bedingungen.

1.5.1 IF..THEN..ELSE-Blockstruktur

Außer mehrzeiligen Bedingungen stellt uns QuickBASIC IF-Blockanweisungen zur Verfügung. Blöcke kennen Sie bereits von WHILE..WEND und FOR..NEXT. Blöcke sind immer durch Schlüsselwörter gekennzeichnet, eines am Blockanfang, ein weiteres am Blockende. Ein Block kann beliebig viele Anweisungen enthalten.

```
i=1
While i<>100
  Print i
  i = i+1
Wend
```

```
For i=1 to 100
  Print i
Next i
```

Auch IF gibt es nun als Blockanweisung. Zur Unterscheidung von der normalen IF-Anweisung muß sich THEN am Schluß der Zeile befinden.

Struktur bedingter Blöcke:

```
IF Bedingung THEN
  Anweisung
  ...
  Anweisung
ELSE
  Anweisung
  ...
  Anweisung
END IF
```

THEN am Zeilenende bedeutet für QuickBASIC, daß nun ein Anweisungsblock beginnt. Die gesamte Blockanweisung wird mit END IF abgeschlossen.

Der Block zwischen IF und END IF kann aus mehreren »Unterblöcken« bestehen. Das Strukturschema zeigt, daß nicht nur der THEN-Zweig einen Anweisungsblock bilden kann, sondern auch der (optionale) ELSE-Zweig.

Mit der IF-Blockanweisung wird unser Ausgangsprogramm erheblich besser lesbar (Programm IF_BLOCK.BAS im Verzeichnis DEMOS).

```
If a=1 Then
  For i=1 to 10
    Print "a besitzt den Wert 1"
  Next i
Else
  For i=1 to 10
    Print "a besitzt einen Wert ungleich 1"
  Next i
End If
```

1.5.2 ELSEIF: Auswahl unter alternativen Bedingungen

QuickBASIC erlaubt mit der Anweisung ELSEIF eine Art »Fallunterscheidung«. Eine Fallunterscheidung ist unter anderem bei der Berechnung der Mehrwertsteuer nötig. Drei Fälle sind denkbar:

1. Es muß keine Mehrwertsteuer berechnet werden.
2. 7% Mehrwertsteuer sind zu berechnen (halber Mehrwertsteuersatz).
3. 14% Mehrwertsteuer sind zu berechnen (voller Satz).

Im folgenden Beispiel wird der Benutzer zuerst nach einem Betrag und anschließend nach dem anzuwendenden Mehrwertsteuersatz gefragt (halber Satz, voller Satz, keine Mehrwertsteuer). Danach wird die resultierende Mehrwertsteuer und der Gesamtbetrag (Ausgangsbetrag plus Mehrwertsteuer) berechnet. Die »konventionelle« Lösung:

```
Input "Ausgangsbetrag"; Betrag

Print "1 = Halber Satz (7%)"          'Benutzer-Informationen
Print "2 = Voller Satz (14%)"         'ausgeben
Print "3 = Keine Mehrwertsteuer"
Input "Bitte Zahl eingeben"; Zahl     'MWSt-Satz erfragen

If Zahl = 1 Then                      'Berechnungen bei halbem
  Print "MWSt =" Betrag/100*7         'MWSt-Satz
  Print "Betrag + MWSt =" Betrag+Betrag/100*7
End If

If Zahl = 2 Then                      'Berechnungen bei vollem
  Print "MWSt =" Betrag/100*14        'MWSt-Satz
  Print "Betrag + MWSt =" Betrag+Betrag/100*14
End If
```

```
If Zahl = 3 Then                    'Berechnungen, wenn keine
  Print "MWSt =" Betrag             'MWSt anzuwenden ist
  Print "Betrag + MWSt =" Betrag
End If
```

QuickBASIC vereinfacht diese umständliche Lösung durch die Anweisung ELSEIF. Mit ELSEIF ist die Prüfung alternativer Bedingungen möglich.

Die Struktur der ELSEIF-Anweisung:

```
IF Bedingung THEN          'IF-Block
 Anweisung
 ...
 Anweisung
ELSEIF Bedingung THEN      'ELSEIF-Block
 Anweisung
 ...
 Anweisung
...
...
ELSEIF Bedingung THEN      'ELSEIF-Block
 Anweisung
 ...
 Anweisung
END IF
```

Das Schema zeigt, daß ebenso wie IF und ELSE auch ELSEIF ein Block von Anweisungen folgen kann. Voraussetzung ist nur, daß sich auch diesmal THEN am Zeilenende befindet.

ELSEIF darf nur in Blockstrukturen verwendet werden! Das heißt, THEN muß das letzte Wort der IF-Anweisung sein. Daher ist das folgende Programm fehlerhaft. Es hält sich nicht an diese Vorschrift. THEN folgen weitere Anweisungen. Der Compiler erkennt nicht, daß wir die Blockstruktur verwenden. Bei der Kompilation wird ELSEIF als »unbekannte Anweisung« abgewiesen.

Falsch

```
If a=Ø Then Print "Hallo"            'Falsch, da keine
ElseIf a=1 Then Print "Guten Tag"    'Blockstruktur (nach THEN
End If                               'folgen weitere Anweisungen)
```

Richtig

```
If a=Ø Then                          'Richtig, da QuickBASIC die
  Print "Hallo"                      'Blockstruktur erkennt.
ElseIf a=1 Then Print "Guten Tag"    'ELSEIF ist nun eine zu-
End If                               'laessige Anweisung.
```

Im Gegensatz zum vorigen Beispiel verwendet das folgende einen ELSEIF-Block. Wie erläutert muß sich THEN am Zeilenende befinden.

```
If a=0 Then                 'Beginn eines Blocks
  Print "Hallo"
ElseIf a=1 Then             'Ebenfalls Beginn eines Blocks
  Print "Guten Tag"
  Print "Guten Abend"
End If                      'Ende des Gesamtblocks
```

ELSEIF eignet sich hervorragend für unser Mehrwertsteuer-Problem. Die erste Bedingung geben wir hinter IF an, die beiden anderen Bedingungen werden durch je einen ELSEIF-Teil behandelt:

Input "Ausgangsbetrag"; Betrag

```
Print "1 = Halber Satz (7%)"          'Benutzerinformationen
Print "2 = Voller Satz (14%)"         'ausgeben
Print "3 = Keine Mehrwertsteuer"
Input "Bitte Zahl eingeben"; Zahl      'MWST-Satz erfragen

If Zahl = 1 Then                       'Halber Satz
  Print "MWST =" Betrag / 100 * 7
  Print "Betrag + MWST =" Betrag + Betrag / 100 * 7
ElseIf Zahl = 2 Then                   'Voller Satz
  Print "MWST =" Betrag / 100 * 14
  Print "Betrag + MWST =" Betrag + Betrag / 100 * 14
ElseIf Zahl = 3 Then                   'Keine MWST
  Print "MWST =" Betrag
  Print "Betrag + MWST =" Betrag
End If
```

Dieses Programm finden Sie auf der Diskette zum Buch im Verzeichnis DEMOS unter dem Namen MWST.BAS.

Zusammenfassung:

1. THEN als letztes Wort (ausgenommen Kommentare) leitet eine Blockanweisung ein. In einer Blockanweisung kann sowohl der THEN- als auch der ELSE-Zweig einen Block von Anweisungen enthalten.
2. In einer Blockanweisung kann mit einer Folge von »ELSEIF Bedingung THEN Anweisung« unter mehreren Bedingungen gewählt werden. (Fallunterscheidung).
3. Soll einer ELSEIF-Anweisung ein Block folgen, muß THEN wiederum das letzte Wort der Zeile sein.
4. Die gesamte Blockanweisung endet mit END IF.

1.6 Benutzerdefinierte Funktionen

Bestimmt kennen Sie die Anweisung DEF FN, die eine Funktion definiert.

Mit DEF FN definieren Sie eine Funktion. Jede Programmiersprache besitzt eine Unmenge bereits »eingebauter« Funktionen. Eine Funktion übergibt in Abhängigkeit von den übergebenen »Argumenten« ein Resultat, einen Wert. Dieser Wert kann eine Zahl oder eine Zeichenkette sein. Beispiele für eingebaute Funktionen sind die Sinus-Funktion (SIN(Argument)), der Absolutbetrag (ABS(Argument)) oder die Stringfunktion RIGHT$(Argument1, Argument2).

Funktionen können Sie als Argumente Konstanten, Variablen oder auch gemischte Ausdrücke übergeben:

- Print Sin(10)
- X = 10: Print Sin(10)
- X = 10: Print Sin(2 * 10)

DEF FN gibt Ihnen die Möglichkeit, eigene Funktionen zu konstruieren. Um eine Funktion zu definieren, die den Inhalt eines Rechtecks berechnet, geben Sie in GW-BASIC an:

```
100 DEF FN RECHTECK(BREITE, LAENGE) = BREITE * LAENGE
110 PRINT FN RECHTECK(10, 20)
```

Mit DEF FN leiten Sie die Funktionsdefinition ein. Dann folgen der Funktionsname und die Argumente. Auf der rechten Seite der Definition befindet sich ein Ausdruck, der das Funktionsergebnis ermittelt.

Diese selbstdefinierte Funktion können Sie jederzeit über den angegebenen Namen »aufrufen« und das Ergebnis weiterverwenden. Sie müssen das Ergebnis sogar weiterverwenden! Ein Funktionsaufruf ist keine Anweisung wie PRINT oder INPUT. Daher müssen Sie QuickBASIC mit einer Anweisung mitteilen, was mit dem Funktionsergebnis geschehen soll. Die Programmzeile

```
120 FN RECHTECK(10, 20)
```

ist unzulässig. Die aufgerufene Funktion übergibt einen Wert; es fehlt jedoch die Anweisung, diesen Wert zum Beispiel auszudrucken (*PRINT FN RECHTECK(10, 20)*) oder einer Variablen zuzuweisen (*X = FN RECHTECK(10, 20)*). QuickBASIC weiß nicht, was mit diesem Wert geschehen soll!

GW-BASIC beschränkt die Definition einer eigenen Funktion auf eine Programmzeile. Damit ist in der Praxis sehr wenig anzufangen. In QuickBASIC

dürfen Funktionen beliebig lang sein. Die Anweisung END DEF kennzeichnet das Ende der Definition.

```
DEF FN Funktionsname (Argumentliste)
  Anweisung 1
  Anweisung 2
  ...
  ...
  Anweisung N
  FN Funktionsname = Ausdruck
END DEF
```

Das Resultat der Funktion bestimmen Sie mit FN Funktionsname = Ausdruck. Diese »Wertzuweisung« muß sich keineswegs wie im Schema am Ende der Definition befinden. Sie darf an beliebigen Stellen innerhalb der Funktion auftauchen, und zwar mehrfach. Den »Datentyp« des Resultats bestimmt die in der Argumentliste angegebene Variable. Ist es eine Stringvariable, ist auch das Ergebnis vom Typ String (eine Zeichenkette). Ist es eine Integervariable, wird ein Integerwert übergeben.

```
Def Fn Rechteck(Breite, Laenge)
  Fn Rechteck = Breite * Laenge
End Def

Print Fn Rechteck(1Ø, 2Ø)
```

Wie zuvor in GW-BASIC wird in QuickBASIC eine Funktion RECHTECK definiert. Beim folgenden Aufruf übergibt das Programm die Parameter 10 und 20. Die Funktion speichert beide Parameter in den Variablen <Breite> und <Laenge>. <Breite> und <Laenge> werden multipliziert. Das Ergebnis – der Inhalt des Rechtecks – ist jener Wert, den die Funktion dem »aufrufenden Programm« übergibt.

Bei der Verwendung von Funktionen sind einige Regeln zu beachten. Vor dem Aufruf muß eine Funktion bereits definiert sein. Außerdem darf sich eine Funktionsdefinition auf keinen Fall innerhalb einer der folgenden Anweisungen befinden:

- SUB..END SUB
- IF..THEN..ELSE
- FOR..NEXT
- WHILE..WEND

Die einfachste Möglichkeit, all diese Regeln einzuhalten: Sie definieren alle benötigten Funktionen gleich am Programmanfang. Erst nach dem Block »Funktionsdefinitionen« kommt Ihr eigentliches Hauptprogramm.

```
1. Funktionsblock
   Def Fn Name1(Argumentliste)
     ...
     ...
   End Def

   Def Fn Name1(Argumentliste)
     ...
     ...
   End Def

2. Hauptprogramm
   ...
   ...
```

In unserem Beispiel verwendete die Funktion RECHTECK die Variablen <Breite> und <Laenge>, um die übergebenen Argumente weiterzuverarbeiten.

Alle Variablen in der Argumentliste sind sogenannte »lokale Variablen«. Wir werden noch sehr intensiv über diese Variablenart diskutieren. Vorläufig nur soviel: Die lokalen Variablen in der Argumentliste beeinflussen niemals Variablen des Hauptprogramms, auch wenn der Variablenname identisch ist!

```
Def Fn Test(X)
  Fn Test = 1ØØ
  X = 1
End Def

X = 12345
Print Fn Test(X), X
```

Programmlauf: `1ØØ  12345`

Diese Funktion ist natürlich völlig sinnlos. Sie liefert immer das Ergebnis 1, egal, welches Argument <X> wir übergeben. Aber sie zeigt, daß das <X> in der Argumentliste die Variable <X> des aufrufenden Programms nicht beeinflußt. <X> besitzt auch nach dem Aufruf der Funktion noch seinen ursprünglichen Wert 12345.

Vorsicht: Nur die Variablen in der Argumentliste sind »lokal«.

```
Def Fn Test(Y)
  Fn Test = 1ØØ
  X = 1
End Def

X = 12345
Print Fn Test(Y), X
```

Programmlauf: `1ØØ  1`

Diesmal ist <X> nicht in der Argumentliste enthalten und daher auch nicht »lokal«. Nach dem Aufruf der Funktion hat <X> einen anderen Wert als

vorher. Dieses Problem kennt jeder BASIC-Programmierer von Unterprogrammen (versehentliches Verändern von Hauptprogrammvariablen im Unterprogramm).

1.6.1 Praktische Anwendung von Funktionen

Die vorigen Beispiele dienten nur dazu, die Theorie anschaulicher zu machen. In der Praxis definieren Sie sicher keine Funktion RECHTECK. Sie macht einfach »zu wenig«.

Im Gegensatz dazu soll Ihnen SEARCH zeigen, wie man Funktionen in der Praxis benutzt. Diese Funktion besitzt eine ähnliche Aufgabe wie die INSTR-Funktion. INSTR prüft, ob eine Zeichenkette in einer anderen enthalten ist.

SEARCH prüft, ob eines der Zeichen einer Zeichenkette in einer anderen Zeichenkette enthalten ist. Wir übergeben SEARCH drei Argumente:

- <S$>: Die zu durchsuchende Zeichenkette
- <Char$>: Der String mit den Zeichen, die uns interessieren
- <Ptr>: Eine ganze Zahl, die angibt, ab welcher Position <S$> durchsucht wird.

Wenn wir SEARCH so aufrufen: *Print Fn Search("Funktionen", "on", 4),* soll die Funktion prüfen, wo in der Zeichenkette »Funktionen« zum ersten Mal eines der Zeichen »o« oder »n« enthalten ist. Und zwar soll diese Prüfung ab dem vierten Zeichen von »Funktionen« stattfinden. Die drei ersten Zeichen (»Fun«) interessieren uns nicht. Nach diesem Aufruf übergibt uns SEARCH – hoffentlich – als Resultat den Wert 7, denn das siebte Zeichen ist ein »o«.

Definieren wir zuerst den »Funktionskopf«, den Namen der Funktion und die Argumentliste:

```
Def Fn Search(S$, Char$, Ptr)
```

Nun brauchen wir eine Schleife, die <S$> ab der Position <Ptr> durchsucht und prüft, ob das untersuchte Zeichen in <Char$> enthalten ist.

```
For i=Ptr to Len(S$)
  If Instr(Char$, Mid$(S$, i, 1)) <> Ø Then FnSearch = i
Next i
```

Mid$(S$, i, 1) »schnappt« sich das aktuelle Zeichen Nummer <i> und INSTR prüft, ob es in <Char$> enthalten ist. Wenn ja, steht das Funktionsergebnis fest. Es ist die aktuelle Position <i> (FnSearch = i). Die Definition endet mit der Anweisung END DEF. Die komplette Funktion SEARCH:

```
Def Fn Search(S$, Char$, Ptr)
  For i=Ptr to Len(S$)
    If Instr(Char$, Mid$(S$, i, 1)) <> Ø Then FnSearch = i
  Next i
End Def


Print Fn Search("Dies ist ein Test", "is", 1)
```

Dieses Programm (Funktion plus aufrufendes Programm) finden Sie unter dem Namen USERDEF1. Aber leider arbeitet es falsch! Laut Aufruf soll die Funktion ermitteln, wo ab Position 1 zum ersten Mal eines der Zeichen »i« oder »s« in der Zeichenkette »Dies ist ein Test« enthalten ist.

Sie könnten auf Anhieb sagen, an Position 2 (das Zeichen »i«). Unsere Funktion ist leider noch recht unfähig. Sie behauptet, ab Position 16 würde zum ersten Mal eines der beiden Zeichen auftreten! Probieren Sie es aus.

Der Grund: In der Schleife ist die Bedingung *If Instr(Char$, Mid$(S$, i, 1)) <> 0* tatsächlich erfüllt, wenn das zweite Zeichen untersucht wird. Der Funktion wird der Wert 2 zugewiesen, der aktuelle Inhalt von <i> (FnSearch = i).

Aber leider ist die Schleife noch nicht zu Ende. Alle Zeichen bis zum Stringende werden gesucht. Jedesmal, wenn eines der Zeichen »i« oder »s« entdeckt wird, erhält die Funktion einen höheren Wert, nämlich immer den gerade aktuellen Wert von <i>. Das Resultat: Die Funktion gibt uns nicht an, wo »i« oder »s« zum ersten Mal auftreten, sondern wo zum letzten Mal eines dieser Zeichen enthalten ist.

Dieses Problem könnten wir mit einem »unfeinen« Sprung aus der Schleife heraus lösen. QuickBASIC kennt jedoch bessere Mittel. Mit der Anweisung EXIT DEF können Sie eine Funktion jederzeit beenden. Genau das tun wir, wenn eines der gesuchten Zeichen entdeckt wird.

```
'       Funktion Search
'       ---------------
'       Funktion: Durchsucht 'S$' ab 'Ptr' und gibt an, ab
'                 welcher Position eines der in 'Char$'
'                 enthaltenen Zeichen auftritt.
'       Hin     : S$    : String
'                 Char$ : Zu suchende Zeichen
'                 Ptr   : Suchstart
'       Zurück  : Positionsnummer (Ø = Suche war erfolglos)

Def Fn Search(S$, Char$, Ptr)
  For i=Ptr to Len(S$)
    If Instr(Char$, Mid$(S$, i, 1)) <> Ø Then Fn Search = i: Exit Def
  Next i
End Def

Print Fn Search("Dies ist ein Test", "is", 1)
```

Dieses Programm finden Sie unter dem Namen USERDEF2 auf der Diskette zum Buch. Der einzige Unterschied zur vorigen Version ist die zusätzliche Anweisung EXIT DEF innerhalb der Schleife. Und diese Version liefert uns das korrekte Resultat 2.

Übrigens: Funktionen sollten Sie möglichst nach dieser »Vorgabe« aufbauen:

- Funktionsbeschreibung
- Beschreibung der von der Funktion erwarteten Argumente
- Beschreibung des übergebenen Resultats

So leid es mir tut: Zum Abschluß dieses Kapitels muß ich Sie leider vor den benutzerdefinierten Funktionen warnen. Eine fantastische Eigenschaft von QuickBASIC sind die »User-Libraries«, einer der Schwerpunkte dieses Buches. User-Libraries gestatten den Aufbau einer Art »Programmbibliothek«. Diese Bibliothek enthält all jene Unterprogramme, die Sie immer wieder benötigen, Eingaberoutinen oder eine Sammlung von Routinen zur Stringbehandlung.

Der Vorteil: Programme in User-Libraries stehen Ihnen unmittelbar – ohne Kompilieren (!) – zur Verfügung. Nach jeder Änderung kompiliert wird nur das aufrufende Hauptprogramm.

Aber, und das ist leider ein gewichtiges »aber«: Obwohl es nicht explizit im Handbuch steht, dürfen User-Libraries keine benutzerdefinierten Funktionen enthalten!

Es ist nicht möglich, von einem Hauptprogramm aus benutzerdefinierte Funktionen aufzurufen, die sich in einer User-Library befinden. Sie erhalten die Fehlermeldung »Funktion nicht definiert«. Und aus diesem Grund werde ich im folgenden auf den Einsatz von benutzerdefinierten Funktionen verzichten. Sie sind zwar sehr nützlich, aber User-Libraries sind ungleich wertvoller.

2

Prozeduren

Das QuickBASIC-Handbuch unterscheidet zwischen »Unterroutinen« und »Unterprogrammen«. Programmteile, die mit GOSUB aufgerufen werden und mit RETURN enden, sind »Unterroutinen«. Teile, die mit dem Wort SUB beginnen und mit END SUB enden, sind »Unterprogramme«.

Diese verwirrende Bezeichnungsweise werde ich nicht übernehmen. Für jeden BASIC-Programmierer ist ein Programmteil, der mit RETURN endet, ein Unterprogramm, keine »Unterroutine«. In diesem Buch kennzeichnet GOSUB..RETURN auch weiterhin ein Unterprogramm. Abschnitte, die mit SUB beginnen und mit END SUB enden, sind »Prozeduren«.

Bevor wir uns intensiv mit Prozeduren beschäftigen, benötigen Sie einiges an »Grundlagenwissen«. Sie werden erfahren, wie QuickBASIC den Hauptspeicher verwaltet, was der »Stack« ist, und was genau beim »Aufruf« einer Prozedur passiert.

Aber vorher sollen Sie wissen, was eigentlich mit dem Ausdruck »Prozedur« gemeint ist. Sie werden sehen, daß Prozeduren gegenüber den herkömmlichen Unterprogrammen enorme Vorteile besitzen.

2.1 Aufbau und Aufruf einer Prozedur

Ebenso wie ein Unterprogramm kann auch eine Prozedur von beliebigen Stellen des Hauptprogramms aus aufgerufen werden.

Prozeduren beginnen mit der Anweisung SUB (=Subprogram) und enden mit END SUB (=End of Subprogram).

Struktur einer Prozedur

```
SUB Prozedurname STATIC
  Anweisung 1
  Anweisung 2
  ...
  Anweisung N
END SUB
```

In der aktuellen QuickBASIC-Version (Version 2.01) muß dem Prozedurnamen das Wort STATIC folgen!

Ebenso wie ein Unterprogramm kann eine Prozedur beliebig viele Anweisungen enthalten, die nach dem Aufruf ausgeführt werden.

Jede Prozedur besitzt einen Namen. Der Aufruf erfolgt mit der Anweisung CALL unter Angabe des Prozedurnamens.

```
CALL Prozedurname
```

Das folgende Demoprogramm zeigt die Ähnlichkeiten von Prozeduren und Unterprogrammen. Beide Versionen – die Prozedur und das Unterprogramm – geben die Zahlen 1-3 aus.

```
'Hauptprogramm
Call Demo
End

'Prozedur
Sub Demo Static
For i=1 to 3
  Print i;
Next i
End Sub
```

```
'Hauptprogramm
Gosub Demo
End

'Unterprogramm
Demo:
For i=1 to 3
  Print i;
Next i
Return
```

Programmlauf:

```
1  2  3
```

```
1  2  3
```

Noch können Sie keine Vorteile von Prozeduren gegenüber Unterprogrammen erkennen. Die Programmfunktion ist in beiden Fällen identisch, die Programmlänge ebenfalls.

Die Unterschiede werden deutlich, wenn wir beide Programme ein wenig modifizieren. Die Zahlen 1-3 sollen nun zweimal ausgegeben werden. Die Prozedur beziehungsweise das Unterprogramm wird in einer Schleife mehrmals aufgerufen.

```
'Hauptprogramm
For i=1 to 2
  Call Demo
Next i
End

'Prozedur
Sub Demo Static
For i=1 to 3
  Print i
Next i
End Sub
```

```
'Hauptprogramm
For i=1 to 2
  Gosub Demo
Next i
End

'Unterprogramm
Demo:
For i=1 to 3
  Print i
Next i
Return
```

Programmausgaben:

```
1 2 3 1 2 3                    1 2 3
```

Die Unterprogramm-Version arbeitet fehlerhaft! Im Unterprogramm wird die Schleifenvariable <I> verändert. Am Ende des ersten Schleifendurchgangs sollte <I> den Wert eins besitzen. Die Schleife im aufgerufenen Unterprogramm weist <I> jedoch zuerst den Wert eins zu und erhöht <I> nach jedem Durchgang. Nach der Rückkehr ins Hauptprogramm besitzt <I> den Wert vier. Da dieser Wert höher ist als der angegebene Endwert zwei, wird die Schleife verlassen.

Dieser Fehler ist schon fast typisch für BASIC-Programme. Hauptprogramm-Variablen können durch den Aufruf von Unterprogrammen unbeabsichtigt verändert werden. Jeder BASIC-Programmierer weiß, wie sehr dadurch die Erstellung größerer Programme erschwert wird.

Interessanterweise tritt dieser Fehler in der Prozedur-Version nicht auf. Offenbar wird die Variable <I> der Hauptprogrammschleife durch die Verwendung von <I> in der Prozedur nicht beeinflußt. Der einzig mögliche Schluß ist merkwürdig, aber dennoch richtig: <I> ist nicht gleich <I>. Die im Hauptprogramm verwendete Variable <I> ist offenbar eine andere Variable als das im Unterprogramm verwendete <I>!

Um dieses Phänomen zu verstehen, ist ein kleiner »Ausflug« nötig. Bevor wir uns weiter mit Prozeduren beschäftigen, zeige ich Ihnen, welche Unterschiede es zwischen Interpretern wie GW-BASIC und Compilern wie QuickBASIC gibt. Vor allem werde ich erläutern, was beim Aufruf einer Prozedur passiert. Danach wissen Sie, warum <I> nicht unbedingt gleich <I> ist.

2.2 Interpreter- und Compilersprachen

Höhere Programmiersprachen wie BASIC besitzen den Vorteil, daß der Programmierer sich nicht um die Speicherverwaltung kümmern muß. Es ist Aufgabe des BASIC-Interpreters oder -Compilers, Zeichenketten und Zahlen in freien Bereichen des Rechnerspeichers abzulegen. Der Programmierer muß nur den Variablennamen kennen, um jederzeit auf die Daten zugreifen zu können.

Mit einem Compiler wie QuickBASIC sollte man sich dennoch eingehender beschäftigen. Viele Fehler sind vermeidbar, wenn man weiß, wie ein Compiler Variablen behandelt.

Speicherzelle zu schreiben. QuickBASIC setzt *X=10* in die Maschinensprache-Anweisung um:

»Schreibe den Wert 10 in die Speicherzelle Nummer 1000«.

Also immer dann, wenn <X> ein neuer Wert zugewiesen wird, überschreibt das Programm einfach den Inhalt der Speicherzelle Nummer 1000. Im Maschinensprache-Programm steht überall die »Hausnummer« 1000, wo sich im Quelltext der Variablenname <X> befand.

2.4 Lokale Variablen

Sie wissen nun, warum QuickBASIC so »quick« ist. Weder für BASIC-Anweisungen noch für den Zugriff auf Variablen ist die langwierige Sucherei in einer Tabelle nötig. Der Prozessor kann die Anweisungen des erzeugten Maschinensprache-Programms sofort ausführen, ohne daß ihm ein Interpreter Anweisung für Anweisung übersetzen muß. Das gesamte Programm ist bereits übersetzt! Hauptprogramm-Variablen müssen ebenfalls nicht gesucht werden. Jeder Variablen wurde bei der Kompilierung ein bestimmter Speicherbereich zugewiesen, und im erzeugten Maschinensprache-Programm befindet sich anstelle des Variablennamens bereits die zugehörige Adresse.

Die beschriebene Art der Variablenverwaltung im Datenbereich gilt allerdings nur für Haupt- und Unterprogramme, nicht für Prozeduren! QuickBASIC greift zur Verwaltung von Prozedur-Variablen auf eine andere Methode zurück.

Prozedur-Variablen sind sogenannte »lokale Variablen« und werden in einem ganz anderen Bereich des Speichers verwaltet, im sogenannten »Stack-« oder auch »Stapel-Speicher«. Prozedur-Variablen sind »dynamisch«. Sie besitzen keine festen, unveränderlichen Adressen, sondern werden beim Aufruf der Prozedur angelegt und bei der Rückkehr aus der Prozedur wieder gelöscht.

Stellen Sie sich unter dem Stapel bitte diesmal kein Hochhaus vor, sondern einen der in vielen Autos anzutreffenden Münzspeicher für Parkuhren. Dieser Vergleich kommt der Wirklichkeit recht nahe. Auf diesen Münzspeicher kann eine Zehnpfennig-Münze geschoben werden. Die Zehnpfennig-Münze ist nun »gespeichert«. Genau so werden auch Prozedur-Variablen gespeichert. Wie eine Zehnpfennig-Münze werden beim Aufruf einer Prozedur die Prozedur-Variablen von oben »auf den Stapel geschoben«.

Die Prozedur kennt die Adressen der von ihr benutzten Variablen in diesem Stapel-Speicher, sie kann auf diese Variablen lesend (*PRINT A$*) und schrei-

bend (*A$="Hallo"*) zugreifen. Die Anweisung SUB END beendet eine Prozedur. Die Prozedur-Variablen werden nun wieder vom Stapel entfernt, gewissermaßen »heruntergezogen«. Das bedeutet, daß der ursprüngliche Stapel-Zustand wieder hergestellt ist. Die Variablen, die die Prozedur verwendete, existieren nicht mehr!

Das heißt, Prozedur-Variablen sind nicht »statisch« wie Hauptprogramm-Variablen. Wie wir zuvor sahen, besitzt jede Hauptprogramm-Variable eine feste Adresse. An dieser Adresse ist während des gesamten Programmlaufs der Variableninhalt gespeichert, auch wenn er sich zwischendurch ändert. Im Gegensatz dazu sind Prozedur-Variablen »dynamische« Variablen. Sie existieren nur vorübergehend. Beim Aufruf der Prozedur werden sie angelegt und bei der Rückkehr aus der Prozedur wieder gelöscht.

Beispiel:

```
'Hauptprogramm
I=1Ø
Call Demo
End

'Prozedur
Sub Demo Static
  I=2Ø
End Sub
```

Die Variable <I> des Hauptprogramms besitzt eine feste Adresse im Datenbereich. Dagegen werden alle in der Prozedur verwendeten Variablen erst beim Aufruf mit CALL im Stapel-Bereich angelegt. In diesem Beispiel verwendet die Prozedur nur eine Variable, <I>. <I> wird beim Aufruf auf den Stapel »geschoben«. Es existieren nun zwei Variablen mit dem Namen <I>: die »statische« Hauptprogramm-Variable <I>, die sich im Datenbereich befindet, und die »dynamische« Prozedur-Variable <I> ganz oben auf dem Stapel.

Und nun der eigentliche Kernpunkt: Ein Hauptprogramm verwendet ausschließlich (!) seine eigenen statischen Variablen und jede Prozedur ebenfalls ausschließlich (!) ihre eigenen dynamischen Variablen.

Man könnte es so ausdrücken: Die Prozedur »weiß« nichts von den Variablen, die das Hauptprogramm verwendet und umgekehrt. Hauptprogramm und Prozeduren benutzen eigene, voneinander unabhängige Variablen!

Die beiden Variablen <I> unseres Beispiels sind also völlig unabhängig voneinander, obwohl sie den gleichen Namen besitzen! Im Hauptprogramm bezeichnet <I> eine Adresse im Datenbereich, in der Prozedur eine völlig andere Adresse im Stapel-Bereich.

Eine Prozedur ist also eine Art »Programm im Programm«. Prozeduren sind nicht einfach Unterprogramme, sondern eigenständige und vom Hauptprogramm unabhängige Programmteile, die ihre eigenen Variablen verwenden.

Erinnern Sie sich an unser Ausgangsprogramm? Das Hauptprogramm verwendete <I>, die aufgerufene Prozedur ebenfalls. Dennoch wurde das »Hauptprogramm-<I>« durch die Prozedur nicht beeinflußt. Unsere Schlußfolgerung damals: Es existieren offenbar zwei voneinander unabhängige Variablen, die zwar den gleichen Namen besitzen, sich aber dennoch nicht beeinflussen.

Inzwischen wissen Sie, daß diese Folgerung richtig war. Beim Aufruf wurde die lokale Variable <I> angelegt. Die Prozedur ändert nur den Wert dieser lokalen Variablen, niemals den Wert der Hauptprogramm-Variablen <I>.

```
'Hauptprogramm
Call Demo
Print "Der Inhalt von I ist:" i
End

'Prozedur
Sub Demo Static
  i=10
End Sub
```

Programmlauf: `Der Inhalt von I ist: 0`

Die Prozedur weist <I> den Wert zehn zu. Dennoch gibt das Hauptprogramm als Inhalt von <I> eine Null aus.

Der Grund: Das in der Prozedur verwendete <I> kennt nur die Prozedur selbst. Das Hauptprogramm verwendet eigene Variablen, es weiß überhaupt

2.5 Übergabe von Variablen

QuickBASIC bietet mit den Prozeduren die Möglichkeit, »modular« zu programmieren. Modular heißt, das Gesamtprogramm wird in einzelne Einheiten unterteilt.

Idealerweise ist ein Modul eine »funktionale Einheit«. Das heißt, jedes Modul besitzt eine fest umrissene Aufgabe und ist ansonsten völlig unabhängig vom restlichen Programm.

Unabhängig vom restlichen Programm sind Prozeduren nun tatsächlich. Sogar zu unabhängig. Wir sahen, daß eine Prozedur eine Art eigenständiges »Programm im Programm« ist mit eigenen Variablen.

Wie sollten Sie nun je eine Prozedur zum Sortieren von Adressen schreiben können, wenn die Prozedur auf diese Adressen – also auf Hauptprogramm-Variablen – nicht zugreifen kann?

Prozeduren sind sinnlos, wenn es keine Möglichkeit des Datenaustauschs zwischen Hauptprogramm und Prozedur gibt. Daher gestattet QuickBASIC diesen Datenaustausch auf Ihren »ausdrücklichen Wunsch« hin. Es gibt die Möglichkeit, einer Prozedur bestimmte Hauptprogramm-Variablen zugänglich zu machen.

Für diese »Variablenübergabe« wird eine Erweiterung der SUB- und der CALL-Anweisung verwendet.

```
CALL Prozedurname (Argumentliste)

SUB Prozedurname (Argumentliste) STATIC
```

»Argumentliste« ist eine Liste jener Variablen, auf die die aufgerufene Prozedur zugreifen soll.

Sie können wählen:

- Soll es der Prozedur nur möglich sein, auf eine Hauptprogramm-Variable »lesend« zuzugreifen (»Variablenübergabe nach Wert«)?
- Oder gestatten Sie der Prozedur auch die Änderung der Hauptprogramm-Variablen (»Übergabe nach Referenz«)?

Der »nur lesende« Zugriff ist zweifellos sicherer, da eine versehentliche Änderung der Hauptprogramm-Variablen nicht möglich ist.

Dennoch werden wir sehr oft die zweite Variante benötigen. Und zwar immer dann, wenn die Prozedur dem Hauptprogramm Ergebnisse mitteilen soll, zum Beispiel das Resultat einer Berechnung.

Beschäftigen wir uns zuerst mit dem »nur lesenden« Zugriff, der Übergabe von Variablen »nach Wert«.

2.5.1 Wertübergabe

Variablen, die ein aufrufendes Programm einer Prozedur übergeben will, müssen in der Argumentliste der SUB-Anweisung mit Kommata getrennt aufgeführt werden.

```
SUB DEMO (A,B) STATIC
```

Die genaue Syntax der CALL-Anweisung hängt von der gewünschten Zugriffsart ab. Soll es der Prozedur nicht möglich sein, den Inhalt der übergebenen Variablen zu verändern, übergeben Sie die Variablen »nach Wert«. In der CALL-Argumentliste wird jede Variable in Klammern gesetzt.

```
CALL DEMO ((A),(B))
```

Die Prozedur DEMO kann nun auf die Inhalte von <A> und <B> zugreifen. »Intern« passiert bei der Übergabe »nach Wert« folgendes:

- Beim Aufruf der Prozedur wird wie bisher eine lokale Prozedur-Variable angelegt.
- Neu ist, daß der Inhalt der übergebenen Hauptprogramm-Variablen in die lokale Variable übertragen wird. Die lokale Variable besitzt nicht den Ausgangswert null, sondern genau den gleichen Inhalt wie die Hauptprogramm-Variable. Der Prozedur wurde »ein Wert übergeben«.

Die Argumentliste kann nicht nur zwei, sondern beliebig viele Variablen enthalten.

Beispiel: `CALL DEMO(A,B,C,D,E,F,G,H)`

Im folgenden Demoprogramm genügt die Übergabe eines Wertes. Die aufgerufene Prozedur berechnet 14 Prozent eines vom Hauptprogramm übergebenen Werts.

```
'Hauptprogramm
Input "Betrag"; X
Call MWSt((X))
End

'Prozedur
Sub MWSt(X) Static
  Print "14% davon:" X*Ø.14
End Sub
```

Programmlauf:
```
Betrag? 2ØØ
14% davon: 28
```

Das Hauptprogramm übergibt der Prozedur MWST den vom Benutzer eingegebenen Betrag, der in <X> enthalten ist. QuickBASIC legt beim Aufruf der Prozedur die lokale Variable <X> an und kopiert in diese lokale Variable den Inhalt der Hauptprogramm-Variablen <X>. Die Prozedur berechnet 14 Prozent von <X> und gibt das Resultat aus. END SUB entfernt die lokale Variable wieder vom Stack.

Da die Variable <X> nach Wert übergeben wird, sollte es der Prozedur nicht möglich sein, die Originalvariable <X> zu verändern. Probieren wir's aus.

```
'Hauptprogramm
Input "Betrag"; X
Call MWSt((X))
Print "Momentaner Wert von X:" X
End

'Prozedur
Sub MWSt(X) Static
  Print "14% davon:" X*Ø.14
  X=1Ø
End Sub
```

Programmlauf:

```
Betrag? 2ØØ
14% davon: 28
Momentaner Wert von X: 2ØØ
```

Die Prozedur verändert zwar <X>. Da die Variable jedoch nach Wert übergeben wurde, ändert sich nur die lokale Variable <X> der Prozedur. Die Hauptprogramm-Variable <X> wird nicht verändert.

Im Demoprogramm wird der Variablenname <X> sowohl vom Hauptprogramm als auch von der Prozedur verwendet. Ebensogut könnte die Prozedur jedoch einen beliebigen anderen Namen verwenden.

```
'Hauptprogramm
Input "Betrag"; X
Call MWSt((X))
Print "Momentaner Wert von X:" X
End

'Prozedur
Sub MWSt(Y) Static
  Print "14% davon:" Y*Ø.14
  Y=1Ø
End Sub
```

Programmlauf:

```
Betrag? 2ØØ
14% davon: 28
Momentaner Wert von X: 2ØØ
```

Die lokale Variable heißt nun <Y>. Wie zuvor wird der Inhalt von <X> in die lokale Variable kopiert, die nun den Namen <Y> besitzt.

Das heißt, in den Argumentlisten der Anweisungen CALL und SUB können völlig verschiedene Variablennamen verwendet werden. Entscheidend ist nur der Variablentyp, der übereinstimmen muß.

Falscher Variablentyp	Richtiger Variablentyp
...	...
Call Demo((X))	Call Demo((X))
End	End
...	...
...	...
Sub Demo(A%) Static	Sub Demo(A) Static
...	...
End Sub	End Sub

Selbstverständlich können Sie auch mehrere Variablen übergeben. Wichtig ist dabei jedoch die Reihenfolge. Der Wert der ersten Variablen in der CALL-Argumentliste wird in die erste Variable der SUB-Argumentliste kopiert, der zweite CALL-Wert in den zweiten SUB-Wert und so weiter.

```
'Hauptprogramm
X=10:Y=20:Z=30
Call Demo((X),(Y),(Z))
End

'Prozedur
Sub (Zahl,Y,Z) Static
  Print Zahl;Y;Z
End Sub
```

Programmlauf: `10 20 30`

Die CALL-Anweisung übergibt die Inhalte von <X>, <Y> und <Z>. Diese drei Werte werden in die lokalen Variablen <Zahl>, <Y> und <Z> kopiert. Daß die Variablennamen verwirrenderweise teils identisch, teils unterschiedlich sind, ist beabsichtigt. Es soll Sie daran erinnern, daß Sie in der SUB-Argumentliste beliebige (!) Namen wählen dürfen. Nicht nur <Zahl> ist eine andere Variable als <X>. Trotz gleicher Namen sind auch die Prozedur-Variablen <Y> und <Z> nicht identisch mit den Hauptprogramm-Variablen <Y> und <Z>.

Übergabe der Werte von:	X	Y	Z
in die lokalen Variablen:	Zahl	Y	Z

Dieses Schema zeigt, wie wichtig die Variablenreihenfolge bei der Übergabe ist. Der Wert der ersten Variable in der CALL-Argumentliste wird immer in der ersten Variablen in der SUB-Argumentliste übergeben und so weiter.

2.5.2 Adreßübergabe (Übergabe nach Referenz)

Die zweite Methode der Variablenübergabe erlaubt es einer Prozedur, den Wert der übergebenen Variablen zu verändern. Diese Übergabeart benötigen Sie immer dann, wenn eine Prozedur Ergebnisse an das Hauptprogramm zurückliefern soll.

Ein Beispiel: Sie schreiben eine Prozedur, die ein beliebiges Array sortieren soll. Das aufrufende Programm übergibt der Prozedur das zu sortierende Array. Nach Rückkehr aus der Prozedur soll das Array keineswegs wie bei der Übergabe nach Wert unverändert sein, sondern nun in sortierter Form vorliegen. Die Prozedur muß die Möglichkeit haben, die übergebenen Variablen zu verändern!

Soll eine Variable »nach Referenz« übergeben werden, entfallen beim Aufruf die Klammern um den Variablennamen.

Nach Wert: `CALL DEMO((A),(B))`
Nach Referenz: `CALL DEMO(A,B)`

Wie zuvor werden der Prozedur DEMO die beiden Variablen <A> und <B> übergeben, diesmal jedoch »nach Referenz«. Den internen Ablauf beschreibe ich an einem Beispiel. Nehmen wir an, die Variable <X> wird einer Prozedur unter dem Namen <A> nach Referenz übergeben.

```
CALL DEMO(X)
...
SUB DEMO(A) STATIC
```

Im Gegensatz zur Übergabe nach Wert wird diesmal nicht der Wert der Originalvariablen in <A> kopiert. Übergabe nach Referenz bedeutet, die Prozedur soll <X> verändern können. Nehmen wir an, der Inhalt von <X> befinde sich in den Speicherzellen ab »Hausnummer« 1000.

QuickBASIC übergibt der Prozedur nicht den Wert von <X>, sondern die Adresse dieser Variablen, also 1000. Die Prozedur weiß nun, daß sich der Inhalt von <A> an Adresse 1000 befindet. Da die Adressen von <X> und <A> identisch sind, könnte man sagen, <X> und <A> »zeigen« auf die gleichen Speicherzellen.

X ----> 1000 <----- A

Der Inhalt von <A> ist wie bei der Übergabe nach Wert mit dem Inhalt von <X> identisch. Diesmal wird jedoch automatisch die Hauptprogramm-Variable <X> verändert, wenn innerhalb der Prozedur <A> ein neuer Wert zugewiesen wird.

```
'Hauptprogramm
X=1Ø
Call Demo(X)                    'Uebergabe von X nach Referenz
Print "Der neue Wert von X ist:" X
End

'Prozedur
Sub Demo(A) Static
  A=2Ø                          'Veraendern der Originalvariablen
End Sub
```

Programmlauf: `Der neue Wert von X ist: 2Ø`

In diesem Beispiel wird <X> nach Referenz übergeben, das heißt, die zusätzliche Klammerung entfällt. Nach der Übergabe der Adresse von <X> zeigt die Prozedur-Variable <A> auf die gleichen Speicherzellen wie <X>.

Die Prozedur weist <A> den Wert 20 zu. Die Speicherzellen ab Nummer 1000 werden mit diesem neuen Wert überschrieben. Da sich ab dieser Adresse auch der Inhalt von <X> befindet, ändert sich gleichzeitig der Inhalt von <X>. Wie der Programmlauf zeigt, besitzt <X> nach Rückkehr aus der Prozedur tatsächlich den neuen Wert 20. Die Prozedur konnte im Gegensatz zu den vorigen Programmbeispielen die Hauptprogramm-Variable beeinflussen.

Wie immer bei der Variablenübergabe spielt der verwendete Name keine Rolle. Wenn Sie wollen, können Sie die Prozedur-Variable ebenfalls <X> nennen.

```
'Hauptprogramm
X=1Ø
Call Demo(X)     'Uebergabe von X nach Referenz
Print "Der neue Wert von X ist:" X
End

'Prozedur
Sub Demo(X) Static
  X=2Ø           'Veraendern der Originalvariablen
End Sub
```

Programmlauf: `Der neue Wert von X ist: 2Ø`

Egal, unter welchem Namen Prozeduren per Referenz übergebene Variablen weiterverarbeiten: Jede Änderung einer Variablen in der SUB-Argumentliste ändert die zugehörige Variable der CALL-Argumentliste, wenn sie per Referenz übergeben wird!

In der Argumentliste können Sie Variablen auch gemischt nach Wert und nach Referenz übergeben. Ein Beispiel: Der Benutzer wird nach einer Zahl <N> gefragt. Eine Prozedur soll die Summe der Zahlen eins bis <N> berechnen (1+2+3+...+N). Das Hauptprogramm übergibt die Variable <N> nach Wert (»nur lesender« Zugriff). Um dem Hauptprogramm die

```
'Hauptprogramm
Dim A$(100)
A$(20)="Hallo"
Call Demo(A$())
Print A$(20)
End

'Prozedur
Sub Demo(XYZ$(1)) Static
  XYZ$(20)="Guten Tag"
End Sub
```

Programmlauf: `Guten Tag`

Unter dem Namen <XYZ$(20)> merkt sich die Prozedur die Adresse der Hauptprogramm-Variablen <A$(20)>. Die Zuweisung

```
XYZ$(20)="Guten Tag"
```

überschreibt den Inhalt der betreffenden Speicherzellen, verändert also die Originalvariable <A$(20)>.

Einzelne Arrayelemente übergeben

Anstelle des gesamten Arrays können Sie auch eine einzige Arrayvariable übergeben. Die leere Klammer in der CALL-Anweisung sagt QuickBASIC, daß ein komplettes Array zu übergeben ist. Eine einzelne Arrayvariable übergeben Sie, indem Sie statt der leeren Klammer den Index der gewünschten Variablen angeben.

```
CALL DEMO(A$(20))
```

In der SUB-Anweisung wird zur Übernahme eine einfache Variable verwendet. Die Angabe der Dimension entfällt.

```
SUB DEMO(T$) STATIC
```

Die Variable <A$(20)> übernimmt die Prozedur unter dem Namen <T$>.

Array-Grenzfunktionen

Alle folgenden Programme befinden sich im Verzeichnis DEMOS (MAXWERT1.BAS, MAXWERT2.BAS und MAXWERT3.BAS).

Angenommen, Sie benötigen eine Prozedur, die den höchsten Wert in einem numerischen Array ermittelt. Die einfachste Möglichkeit:

MaxWert: Version 1 (MAXWERT1.BAS)

```
'Hauptprogramm
Dim A(8Ø)
For i=1 to 8Ø
  A(i)=Rnd(1)
Next i
Call MaxWert(A())                        'Übergabe des Arrays A(..)
End

'Prozedur
Sub MaxWert(Array(1)) Static             'Übernahme in ARRAY(..)
  Max=Ø
  For i=1 to 8Ø
    If Array(i) > Max Then Max=Array(i)
  Next i
  Print "Der grösste Wert ist:" Max
End Sub
```

Das Hauptprogramm ermittelt 80 Zufallszahlen, die im Array <A(..)> gespeichert werden. Die Prozedur MAXWERT soll die größte dieser Zahlen ermitteln und ausgeben. Die Prozedur prüft 80 Zahlen mit einer Schleife.

```
For i=1 to 8Ø
```

Diese Schleife ist schuld daran, daß die Prozedur nur sehr eingeschränkt verwendbar ist. Sie prüft immer (!) 80 Arrayvariablen. Wenn Sie ein anderes Array mit zum Beispiel 200 Variablen aufbauen, können Sie diese Prozedur nicht mehr verwenden. Sie benötigen eine zweite Prozedur, die als Schleifenendwert nicht 80, sondern 200 verwendet.

Soll die Prozedur in der Lage sein, die größte Zahl eines beliebigen (!) Arrays zu ermitteln, müßte man eigentlich außer dem Array selbst auch die Arraygröße angeben.

```
ANZAHL=8Ø                          'Anzahl definieren
CALL MAXWERT(A(),(ANZAHL))         'Array und Anzahl übergeben
...
SUB MAXWERT(ARRAY(1),ANZAHL) STATIC
```

Dank der »Grenzfunktionen« von QuickBASIC können wir uns die Übergabe der Arraygröße ersparen. Die Funktionen LBOUND (Lower bound = untere Grenze) und UBOUND (higher bound = obere Grenze) ermitteln den höchsten beziehungsweise niedrigsten Index eines Arrays. Das Funktionsargument ist jeweils der Arrayname.

```
LBOUND(A%) => ermittelt den niedrigsten Index des Arrays <A%(..)>
UBOUND(A%) => ermittelt den höchsten Index des Arrays <A%(..)>
```

Die »Indexklammer« müssen Sie nicht angeben. Die Funktion LBOUND beziehungsweise UBOUND »weiß«, daß mit dem Argument <A%> das Array <A%(..)> gemeint ist.

UBOUND erlaubt der Prozedur, die Obergrenze des übergebenen Arrays zu ermitteln.

MaxWert: Version 2 (MAXWERT2.BAS)

```
'Hauptprogramm
Dim A(8Ø)
For i=1 to 8Ø
  A(i)=Rnd(1)
Next i
Call MaxWert(A())                        'Übergabe des Arrays A(..)
End

'Prozedur
Sub MaxWert(Array(1)) Static             'Übernahme in ARRAY(..)
  Max=Ø
  For i=1 to UBound(Array)               'Obergrenze ermitteln
    If Array(i) > Max Then Max=Array(i)
  Next i
  Print "Der grösste Wert ist:" Max
End Sub
```

In Version 2 wird der Schleifenendwert durch die Funktion UBound(Array) bestimmt, die im Beispiel den Wert 80 als höchsten Index ermittelt.

Version 3 demonstriert die »Rückübergabe« von Prozedurergebnissen. In dieser Version ermittelt die Prozedur ebenfalls die größte im Array enthaltene Zahl. Diesen Wert gibt diesmal jedoch nicht die Prozedur, sondern das aufrufende Programm auf dem Bildschirm aus.

MaxWert: Version 3 (MAXWERT3.BAS)

```
'Hauptprogramm
Dim A(8Ø)
For i=1 to 8Ø
  A(i)=Rnd(1)
Next i
Call MaxWert(A(),X)                      'Übergabe von A(..) und
Print "Der grösste Wert ist:" X          'Ergebniß-Rückgabe in X
End

'Prozedur
Sub MaxWert(Array(1),Max) Static         'Übernahme in ARRAY(..) und
  Max=Ø                                  'Rückgabe des Wertes von MAX
  For i=1 to UBound(Array)               'Obergrenze ermitteln
    If Array(i) > Max Then Max=Array(i)
  Next i
End Sub
```

Der Aufruf Call Maxwert(A( ),X) übergibt außer dem Array die numerische Variable <X>. <X> soll nach Rückkehr aus der Prozedur das Ergebnis enthalten, die gefundene größte Zahl. Daher muß (!) <X> nach Referenz (ohne Klammerung!) übergeben werden. Die Prozedur kennt nun die Adresse von <X> und bezieht sich bei jeder Änderung von <Max> auf

diese Adresse. Denken Sie immer daran: Es ist völlig unwichtig, ob bei der Übergabe von Werten (Übergabe nach Wert) oder Adressen (Übergabe nach Referenz) mit CALL und dem »Empfang« mit SUB unterschiedliche Variablennamen verwendet werden.

2.6 Globale Variablen

Inzwischen wissen Sie, wie wertvoll lokale Variablen sind. Für jeden BASIC-Programmierer ist es eine Erleichterung, nicht mehr pedantisch darauf achten zu müssen, daß Hauptprogramm und Unterprogramm nicht versehentlich die gleichen Variablen benutzen.

Und wie Sie sahen, ist für eine Prozedur dennoch der Zugriff auf Variablen des aufrufenden Programms möglich. Wenn Sie sehr sorgfältig sind, übergeben Sie Variablen – wann immer möglich – nach Wert. Die Übergabe nach Referenz wenden Sie wirklich nur dann an, wenn die Prozedur dem aufrufenden Programm etwas »mitteilen« muß.

Ich warne Sie gleich: An diese ideale Methode werde ich mich mit meiner etwas »lässigeren« Programmierweise nicht immer halten. Immerhin bedeutet die Übergabe nach Wert die Notwendigkeit zweier zusätzlicher Tastenbetätigungen (*CALL DEMO((A))* statt *CALL DEMO(A)*).

Doch Scherz beiseite: Die Notwendigkeit, einer Prozedur alle benötigten Parameter ausdrücklich in der Argumentliste zu übergeben, ist oft sehr lästig. Unangenehm ist diese Notwendigkeit, wenn Sie in einem Programm immer wieder die gleichen Variablen benötigen.

Ein Beispiel: die normale Farbdarstellung schalten Sie mit *COLOR 7,0* ein und die inverse Darstellung mit *COLOR 0,7*. Invers bedeutet, daß die Zeichen in der Standard-Hintergrundfarbe ausgegeben werden (Farbwert 0), und daß für den Hintergrund die Standard-Zeichenfarbe verwendet wird (Farbwert 7).

Wenn Sie Ihr Programm möglichst lesbar schreiben wollen, kommen Sie vielleicht auf die Idee, die Farbwerte durch aussagekräftige Variablennamen zu ersetzen.

```
CharCol = 7   'Standard-Zeichenfarbe
BackCol = Ø   'Standard-Hintergrundfarbe
```

Um zum Beispiel die inverse Darstellung einzuschalten, schreiben Sie nun statt *COLOR 0,7* einfach *COLOR BackCol,CharCol*. Sie müssen nicht weiter

über die Bedeutung der Farbwerte 0 und 7 nachdenken, da die Variablennamen <CharCol> und <BackCol> eindeutig sind.

Diese Methode hat leider einen Haken. Die beiden »Farbvariablen« definieren Sie natürlich im Hauptprogramm. Aufgerufene Prozeduren wissen jedoch nichts davon – sie kennen diese beiden Hauptprogramm-Variablen nicht.

```
'Hauptprogramm
CharCol=7: BackCol=Ø
Call Demo
End

'Prozedur
Sub Demo Static
  Color BackCol,CharCol
  Print "Hallo"
End Sub
```

Dieses Demoprogramm wird daher die Zeichenkette »Hallo« ganz sicher nicht invers ausgeben. Sie müssen jeder Prozedur, die <CharCol> und <BackCol> verwendet, beide Variablen explizit übergeben.

```
'Hauptprogramm
CharCol=7: BackCol=Ø
Call Demo(CharCol,BackCol)
End

'Prozedur
Sub Demo(CharCol,BackCol) Static
  Color BackCol,CharCol
  Print "Hallo"
End Sub
```

Das heißt, Ihnen steht in einem größeren Programm einiges an ziemlich unsinniger Schreibarbeit bevor. Bei jedem Aufruf einer Prozedur, die die Farbeinstellung mit der COLOR-Anweisung verändert, müssen Sie außer den eigentlich wichtigen Parametern immer wieder unsere beiden »Farbvariablen« übergeben. Ich nehme an, daß Sie ohne eine vernünftige Lösung für dieses Problem ebenso wie ich wieder zur alten Methode zurückkehren (*COLOR 0,7*).

2.6.1 Die Anweisung COMMON

Zum Glück gibt es eine sehr elegante Lösung. Variablen können als »global« deklariert werden. Globale Variablen sind allen Prozeduren ohne explizite Übergabe zugänglich.

Die Deklaration globaler Variablen basiert auf der Anweisung COMMON (=gemeinsam), die auch GW-BASIC besitzt.

COMMON wird auch in GW-BASIC verwendet, um einem mit CHAIN »nachzuladenden« Programm Variablen zu übergeben. Die Dokumentation zu GW-BASIC ist manchmal sehr verwirrend. Meiner Ansicht nach ist die Beschreibung von COMMON fast schon unzumutbar. Wirr durcheinander werden Sie mit dem »Attribut SHARED«, mit »benannten« und »unbenannten« COMMON-Blöcken konfrontiert. Ich werde mich im Folgenden auf jene Zusätze zu COMMON beschränken, die in der Praxis auch wirklich benötigt werden.

Wie gesagt, COMMON übergibt normalerweise Variablen an ein verkettetes Programm:

```
COMMON Variablenliste
```

Beispiel:

```
100 COMMON A,B,X,Y
110 CHAIN "PROG2"
```

Die COMMON-Anweisung in Zeile 100 »schützt« die vier angegebenen Variablen. Die CHAIN-Anweisung lädt PROG2 und startet es (wie LOAD"PROG2" mit anschließender Eingabe von RUN). Der Inhalt der mit COMMON geschützten Variablen bleibt beim Aufruf von PROG2 erhalten. In unserer neuen Ausdrucksweise könnte man sagen: Die vier Variablen werden PROG2 »übergeben«. Sie sind dem nachgeladenen Programm »bekannt«.

COMMON muß sich am Programmanfang befinden, vor allen »ausführbaren« Anweisungen. »Ausführbar« sind alle Anweisungen bis auf:

- COMMON selbst
- DEFTyp
- DIM
- OPTION BASE
- REM
- »METABEFEHLE«

Mein Vorschlag: COMMON-Anweisungen sollten die allerersten Anweisungen in Ihrem Programm sein. Dann werden Sie niemals Schwierigkeiten mit einer falschen Reihenfolge bekommen.

2.6.2 Das Attribut SHARED

Sie sahen, wie COMMON bei verketteten Programmen zur Variablenübergabe eingesetzt wird. Zur Übergabe von Variablen an Prozeduren benötigen wir eine erweiterte Fassung der COMMON-Anweisung:

```
COMMON SHARED Variablenliste
```

COMMON mit dem Attribut SHARED deklariert alle aufgeführten Variablen als »globale« Variablen. Die Anweisung

```
COMMON SHARED A,B
```

deklariert <A> und <B> als global. Der Ausdruck »global« sagt aus, daß eine globale Variable praktisch das Gegenteil einer lokalen ist. Lokale Variablen sind nur der betreffenden Prozedur bekannt. Globale Variablen sind im gesamten Programm bekannt. Der Effekt ist praktisch identisch mit der Übergabe der betreffenden Variablen nach Referenz, betrifft jedoch alle im Programm enthaltenen Prozeduren. Auf globale Variablen kann jede Prozedur lesend und schreibend zugreifen.

COMMON SHARED löst unser Problem mit den »Farbvariablen« <CharCol> und <BackCol>. Wir deklarieren beide Variablen am Programmanfang als global und ersparen uns jede weitere Übergabe.

```
'Hauptprogramm
Common Shared CharCol,BackCol      'CharCol und BackCol sind
                                   'globale Variablen
CharCol=7: BackCol=Ø
Call Demo
End

'Prozedur
Sub Demo Static
  Color BackCol,CharCol
  Print "Hallo"
End Sub
```

<CharCol> und <BackCol> sind nicht nur der Prozedur DEMO, sondern auch allen weiteren im Gesamtprogramm vorhandenen Prozeduren bekannt. Wie erwähnt können lokale Variablen von Prozeduren auch verändert werden (analog der Übergabe nach Referenz):

```
'Hauptprogramm
Common Shared A          'A : globale Variable
A = 1Ø
Call Demo
Print "A hat den Wert:" A

'Prozedur
Sub Demo Static
  A = 2Ø
End Sub
```

Programmlauf: `A hat den Wert: 2Ø`

Dieses Programm finden Sie unter dem Namen COMMON.BAS. Es deklariert <A> als globale Variable. Daher kann die Prozedur DEMO <A> beeinflussen.

Seien Sie also bitte vorsichtig mit der Deklarierung globaler Variablen. Es ist nicht sinnvoll, aus »Schreibfaulheit« nahezu alle in einem Programm vorkommenden Variablen als global zu deklarieren. Sicher, Sie ersparen sich jede weitere Parameterübergabe. Aber dafür verzichten Sie auf alle Vorteile lokaler Variablen. Sie sind wieder so weit wie zuvor. Jede Prozedur kann versehentlich Variablen des Hauptprogramms verändern.

Deklarieren Sie daher bitte wirklich nur jene Variablen als global, die Sie sonst immer wieder an nahezu jede Prozedur übergeben müßten. Nicht nur einfache Variablen, auch Arrays können als global deklariert werden.

```
COMMON SHARED Array(Dimension)
```

In der COMMON SHARED-Anweisung geben Sie den Arraynamen und die Dimension an.

```
'Hauptprogramm
Common Shared A(1)
A(5)=1234
Call Demo
End

'Prozedur
Sub Demo Static
  Print A(5)
End Sub
```

Programmlauf: `1234`

In diesem Programm wird das Array <A(..)> global deklariert und ist daher allen Prozeduren bekannt. Wie der Programmlauf zeigt, kann die Prozedur DEMO tatsächlich auf <A(..)> zugreifen und mit dem Wert 1234 arbeiten, der <A(5)> im Hauptprogramm zugewiesen wurde.

2.6.3 Die Anweisung STATIC

QuickBASIC löscht lokale Variablen, wenn die Prozedur beendet ist. Manchmal ist es jedoch nötig, daß die in der Prozedur verwendeten Variablen beim nächsten Aufruf noch die gleichen Werte besitzen.

Ein Beispiel: In einem Programm gibt es eine Funktion »Zugriffspfad angeben«. Der Benutzer kann ein beliebiges Verzeichnis angeben, zum Beispiel »C:\QB\DEMOPROG«. Mit der Anweisung CHDIR (change directory) wird das aktuelle Verzeichnis gewechselt. Das Programm verwendet nun das vom Benutzer angegebene Verzeichnis.

```
Sub EditPfad Static
  Input "Neues Verzeichnis";Dir$     'nach Verzeichnis fragen
  Chdir Dir$                         'Verzeichnis wechseln
End Sub
```

Richtig komfortabel wird diese Prozedur, wenn der Benutzer vor jedem Wechsel über das momentan eingestellte Verzeichnis informiert wird. Das heißt, der Inhalt von <Dir$> muß bis zum nächsten Aufruf von EDITPFAD erhalten bleiben.

Die einfachste Methode: Sie deklarieren <Dir$> als globale Variable, die nicht nach der Rückkehr aus der Prozedur gelöscht wird.

Der Nachteil: <Dir$> als globale Variable zu deklarieren, hieße, »mit Kanonen auf Spatzen zu schießen«. Es ist etwas übertrieben, eine Variable als global zu deklarieren, die nur in einer einzigen Prozedur benötigt wird.

Vor allem besteht die Gefahr, daß <Dir$> versehentlich in einer anderen Prozedur (oder im Hauptprogramm) geändert wird.

Die optimale Lösung ist die Verwendung der Anweisung STATIC. Verwechseln Sie diese Anweisung nicht mit dem Attribut (!) STATIC, das jeder Unterprogramm-Deklaration folgt!

STATIC bewirkt, daß der Wert der angegebenen lokalen Variablen bis zum nächsten Aufruf der Prozedur erhalten bleibt.

```
Sub EditPfad Static
  Static Dir$                          '<Dir$> bleibt erhalten
  Print "Aktuelles Verzeichnis" Dir$   'momentanen Inhalt zeigen
  Input "Neues Verzeichnis";Dir$       'nach Verzeichnis fragen
  Chdir Dir$                           'Verzeichnis wechseln
End Sub
```

Angenommen, Sie geben »C:\QB« als neues Verzeichnis an. Wenn Sie später das Verzeichnis erneut wechseln, werden Sie zuerst über das aktuelle Verzeichnis informiert (»C:\QB«). <Dir$> besitzt immer noch den alten Inhalt. Und dennoch kann <Dir$> nicht versehentlich vom Hauptprogramm oder einer anderen Prozedur beeinflußt werden (im Unterschied zu globalen Variablen). Man könnte sagen, STATIC »schützt« und bewahrt den Inhalt einer lokalen Variablen.

Wenn Sie mit STATIC den Inhalt eines Arrays schützen wollen, geben Sie einfach den Arraynamen ohne Klammern oder Index an:

```
Print "1.Aufruf": Call Demo
Print "2.Aufruf": Call Demo

Sub Demo Static
  Static A$
  Print A$(1), A$(2)
  A$(1) = "Dies ist ein Test"
  A$(2) = "Dies auch"
End Sub
```

Programmlauf:
```
1.Aufruf
2.Aufruf
Dies ist ein Test
Dies auch
```

Beim ersten Aufruf dieser Prozedur hat <A$(1)> noch keinen Inhalt. In der Prozedur wird der Variablen die Zeichenkette »Dies ist ein Test« zugewiesen. Und da der Inhalt erhalten bleibt, gibt die PRINT-Anweisung beim nächsten Aufruf »Dies ist ein Test« aus. Das Gleiche gilt für <A$(2)> und alle anderen Variablen des »statischen« Arrays.

Übrigens: Die Deklaration einer Variablen als global (mit COMMON SHARED) kann mit der Anweisung STATIC für eine bestimmte Prozedur aufgehoben werden. Das heißt, auch wenn eine Variable mit COMMON SHARED als global deklariert wurde, kann mit STATIC erreicht werden, daß sie für die betreffende Prozedur lokal ist.

Das folgende Programm finden Sie auf der Begleitdiskette unter dem Namen STATIC.BAS.

```
Common Shared A$
A$ = "Test"
Call Demo
Print A$

Sub Demo Static
  Static A$
  Print A$
  A$ = "Kein Test"
  Print A$
End Sub
```

Programmlauf: `Test`

<A$> wird als globale Variable deklariert und erhält den Inhalt »Test«. Dank STATIC ist <A$> in der Prozedur jedoch eine lokale Variable, also nicht identisch mit der Variablen <A$> des Hauptprogramms.

Daher enthält die globale Variable <A$> auch nach Rückkehr aus der Prozedur noch den Inhalt »Test«. Die Zuweisung in der Prozedur TEST betrifft nicht die globale, sondern die Prozedur-Variable <A$>.

Diese Eigenschaft von STATIC ist recht praktisch, wenn Sie in einer Prozedur eine lokale Variable mit einem Namen verwenden wollen, den bereits eine als global deklarierte Variable besitzt.

3

Entwicklung einer Routinen-Sammlung

Den ersten – mehr theoretischen – Teil haben Sie nun hinter sich. Ich hoffe, Sie haben ihn gut überstanden. Nun wird es praktisch. Wir nähern uns den angekündigten Projekten. Zuvor muß ich Sie noch über den $INCLUDE-Befehl und zwei sehr wichtige Dateien auf der Diskette zum Buch informieren.

3.1 Der Metabefehl $INCLUDE

Die verschiedenen Programmprojekte sind praktisch völlig unabhängig voneinander. Es gibt jedoch eine Gemeinsamkeit: Alle Programme benutzen immer wieder die gleichen globalen Variablen, unter anderem die »Farbvariablen« <CharCol> und <BackCol>. Das heißt, am Programmanfang befinden sich immer die gleichen COMMON SHARED-Anweisungen.

Aufbau der Programmprojekte

```
'***** Programm Nummer 1 *****
'* Deklaration der globalen Variablen *
Common Shared CharCol,BackCol
...
...
'* Hauptprogramm *
...
...


'***** Programm Nummer 2 *****
'* Deklaration der globalen Variablen *
Common Shared CharCol,BackCol
...
...
'* Hauptprogramm *
...
...
```

```
...

'***** Programm Nummer3 *****
'* Deklaration der globalen Variablen *
Common Shared CharCol,BackCol
...
...
'* Hauptprogramm *
...
...
```

In jedem Programm werden zuerst die globalen Variablen deklariert, dann erst folgt das eigentliche Programm. Natürlich ist es lästig, bei jedem neuen Projekt zuerst immer den gleichen Vorspann abzutippen. In GW-BASIC würden Sie sich diese Abtipperei mit der MERGE-Anweisung erleichtern.

```
MERGE "Name"
```

MERGE bindet die angegebene ASCII-Datei in das aktuelle Programm ein, »vermischt« es mit ihm. Das heißt, Sie schreiben alle globalen Deklarationen in eine Datei. In jedes neue Programm »mergen« Sie diese Datei ein.

QuickBASIC kennt keinen MERGE-Befehl. Tragisch ist das nicht, denn mit dem »Metabefehl« $INCLUDE ist ein hervorragender Ersatz vorhanden. Sobald QuickBASIC bei der Kompilierung eines Programms auf den $INCLUDE-Befehl stößt, setzt es die Kompilierung mit der angegebenen Datei fort. Anschließend wird die Kompilierung des ursprünglichen Programms nach dem $INCLUDE-Befehl fortgesetzt.

```
REM $INCLUDE: 'Dateiname'
```

Achtung: Alle Metabefehle, also auch $INCLUDE, beginnen mit einem einfachen Anführungszeichen (» ' «) oder einer REM-Anweisung!

Beispiel: Im Speicher befindet sich momentan der folgende Programmtext:

```
'Hauptprogramm
i=10
REM $INCLUDE: 'test'
End
```

Die Datei TEST enthält eine einzige Anweisung:

```
Print i
```

Wenn Sie das erste Programm kompilieren, passiert folgendes:

1. Das Hauptprogramm wird bis zur Anweisung $INCLUDE ganz normal kompiliert.
2. Nun liest QuickBASIC die angegebene Datei ein und setzt die Kompilierung mit den darin enthaltenen Anweisungen fort.

3. Das Programm im Speicher wird weiterkompiliert.

$INCLUDE können Sie verwenden, um allgemeine »Module« in verschiedene Programme einzubinden. Nehmen wir an, Sie haben irgendwann einmal eine Prozedur zur Lösung von Gleichungssystemen entwickelt.

```
'Prozedur
Sub Gleichung Static
  ...
  ...
End Sub
```

Die Prozedur speicherten Sie unter dem Namen GLEICHG. Nun sind Sie gerade an einem Statistik-Programm, das diese Prozedur benötigt. Entweder tippen Sie den gesamten Programmtext der Prozedur ab, oder aber Sie fügen in Ihr Hauptprogramm eine $INCLUDE-Anweisung ein.

```
'Statistik-Programm
...
...
$INCLUDE: 'gleichg'
...
...
```

$INCLUDE ermöglicht Ihnen die Erstellung einer im Lauf der Zeit immer umfangreicher werdenden »Bibliothek« oder »Library«. Jedes »Modul« der Library befindet sich in einer eigenen Datei und wird bei Bedarf mit einem $INCLUDE-Befehl in das Programm eingebunden, an dem Sie gerade arbeiten.

3.2 Deklaration globaler Variablen (COMDEF.BAS)

Mit $INCLUDE werden wir in all unsere Programme eine Datei einbinden, die fast nur aus COMMON SHARED-Anweisungen besteht, also immer wieder benötigte Variablen als global deklariert. Die Datei befindet sich natürlich auch auf der Diskette zum Buch, und zwar unter dem Namen COMDEF.BAS (common definitions = gemeinsame Definitionen).

Außer COMMON SHARED enthält die Datei zwei weitere Anweisungen:

```
DEFINT a-z
OPTION BASE 1
```

DEFINT A–Z deklariert alle Variablen als Integervariablen. QuickBASIC rechnet mit Integervariablen deutlich schneller als mit Variablen einfacher

oder gar doppelter Genauigkeit. Daher ist es sinnvoll, zum Beispiel als Schleifenvariablen immer Integervariablen zu verwenden.

Entweder:	oder einfacher:
	`DefInt a-z`
	`...`
	`...`
`For i%=1 to 100`	`For i=1 to 100`
`  Print "Hallo"`	`  Print "Hallo"`
`Next i%`	`Next i`

Und Sie wissen ja: Spezielle Typdeklarationen im Programm wie <A$>, <X!> oder <SUMME#> sind »stärker« als globale Deklarationen mit DEFTyp. Wenn wir sie brauchen, stehen uns daher alle anderen Variablentypen weiterhin zur Verfügung.

OPTION BASE 1 legt als Untergrenze von Arrays den Index 1 fest (normalerweise 0). Diese Anweisung ist sinnvoll, da der Arrayindex 0 in Programmen praktisch nie benutzt wird.

```
'       ****************************************************
'       *   COMDEF.BAS = Common Definitions meiner Programme  *
'       ****************************************************


DefInt a-z       'Grundtyp numerischer Variablen: INTEGER
Option Base 1    'Arrayuntergrenze: 1


'       ************************************
'       *   Globale Variablen/Konstanten   *
'       ************************************

'       * Tasten-Deklarationen *
Common Shared zEsc$, zRight$, zLeft$, zUp$, zDown$, zReturn$,_
              zHome$, zEnd$, zDel$, zIns$, zBack$, zCtrlRight$,_
              zCtrlLeft$, zCtrlHome$, zCtrlEnd$, zCtrlT$, zCtrlY$

'       * Standardstrings *
Common Shared Alpha$, Num$, jn$, Terminator$

'       * Datentyp Boolean *
Common Shared False, True

'       * Standard-Zeichenattribute *
Common Shared CharCol, GroundCol, Normal, Invers, Intensiv, Insert

'       * Windowvariablen *
Common Shared Scr(1), ScrTab(1), ScrPage
Common Shared Header$, Menue$(1), WOffset
```

In COMDEF.BAS werden fünf Typen von Variablen als global deklariert.

1. Tasten: Ob eine Sondertaste wie CURSOR RIGHT oder INSERT gedrückt wurde, prüft man bei Abfragen mit INKEY$ normalerweise so:

```
A$=Inkey$
If Len(A$)=2 And Right$(A$,1)=82 Then...
```

   Bei gedrückten Sondertasten übergibt INKEY$ einen Zwei-Zeichen-String. Das linke Zeichen enthält CHR$(27), den Code der ESC-Taste, und das rechte Zeichen den eigentlichen Tastencode. In meinen Programmen weise ich alle wichtigen Tastencodes verschiedenen Variablen zu, zum Beispiel mit

```
zIns$=Chr$(82)  'Tastencode der INSERT-Taste: 82
```

   Wenn die Variable <zIns$> global, also allen Prozeduren bekannt ist, werden Tastaturabfragen besser lesbar. Die ohne Tabelle der Tastencodes völlig nichtssagende Anweisung

```
...If Right$(A$,1)=82 Then...
```

   wird in allen Prozeduren ersetzt durch

```
...If Right$(A$,1)=zIns$ Then ...
```

   Diese erhöhte Lesbarkeit bei Tastaturabfragen ist der einzige Sinn des Blocks »Tasten«. Für alle wichtigen Sondertasten werden aussagekräftige globale Variablen verwendet.

2. Standardstrings: »Alpha$«, »Num$« und »jn$« werden von verschiedenen Prozeduren häufig bei Eingabeüberprüfungen benötigt. »Alpha$« wird alle alphanumerischen Tasten enthalten. »Num$« enthält alle numerischen Tasten (also die Zeichen »0123456789.-«). »jn$« ist für Eingaben zuständig, bei denen der Benutzer nur mit »Ja« oder »Nein« antworten darf und enthält die beiden Zeichen »j« und »n«. Auf »Terminator$« werde ich bei passender Gelegenheit eingehen.

3. Datentyp Boolean: BASIC kennt im Gegensatz zu Pascal, Modula oder C keinen Datentyp »Boolean«. Eine »Boolesche Variable« kann nur einen von zwei Zuständen annehmen: »True« oder »False«. Boolesche Variablen sind hervorragend für Flags geeignet. BASIC-Programmierer verwenden für Flags normalerweise die »Zustände« 0 und 1.

```
...If Flag=1 Then...
...If Flag=Ø Then...
```

   Genau diese beiden Werte werden in unseren Programmen den globalen Variablen »True« und »False« zugewiesen. Die Abfrage eines Flags sieht dann so aus:

```
...If Flag=True  Then...
...If Flag=False Then...
```

4. Standard-Zeichenattribute: Dieser Block deklariert außer unseren »Farbvariablen« <CharCol> und <BackCol> noch einige andere Farbwerte als global.

5. Window-Variablen: werden später besprochen.

3.3 Initialisierung globaler Variablen (INIT.BAS)

In COMDEF.BAS werden zwar globale Variablen deklariert, diesen Variablen aber noch keine Werte zugewiesen. Um »True« den Wert 1, »CharCol« den Wert 7 und allen übrigen Variablen ebenfalls »ihre« Werte zuzuweisen, wird eine zweite »$INCLUDE-Datei« verwendet. Die Datei INIT.BAS enthält alle Anweisungen zur »Initialisierung« der globalen Variablen.

```
'       ****************************************************
'       * INIT.BAS = Initialisierung der globalen Variablen *
'       ****************************************************

'       * Tastencodes *
zEsc$       = Chr$(27)        'Esc
zRight$     = Chr$(77)        'Cursor Right
zLeft$      = Chr$(75)        'Cursor left
zUp$        = Chr$(72)        'Cursor up
zDown$      = Chr$(80)        'Cursor down
zReturn$    = Chr$(13)        'Return
zHome$      = Chr$(71)        'Home
zEnd$       = Chr$(79)        'End
zDel$       = Chr$(83)        'Delete
zIns$       = Chr$(82)        'Insert
zBack$      = Chr$(8)         'Backspace
zCtrlRight$ = Chr$(116)       'Kombination Ctrl + Cursor Right
zCtrlLeft$  = Chr$(115)       'Kombination Ctrl + Cursor Left
zCtrlHome$  = Chr$(119)       'Kombination Ctrl + Home
zCtrlEnd$   = Chr$(117)       'Kombination Ctrl + End
zCtrlY$     = Chr$(25)        'Kombination Ctrl + y
zCtrlT$     = Chr$(20)        'Kombination Ctrl + t

'       * Standardstrings *
Alpha$      = "yxcvbnm,.-asdfghjklöäqwertzuiopü+#1234567890ß'"_
"<YXCVBNM;:_ASDFGHJKLÖÄQWERTZUIOPÜ*^!§$%&/()=?'>\ "+Chr$(34)
Num$        = "1234567890.- "
JN$         = "jn"
Terminator$ = ",;.:!§$%&/()=?<>#^+-*_\' " + Chr$(34)
```

```
'        * Datentyp Boolean *
False      = Ø
True       = 1

'        * Standard-Zeichenattribute *
CharCol    = 7       'normale Zeichenfarbe
GroundCol  = Ø       'normale Hintergrundfarbe
Normal     = 7       'Attribut für Normaldarstellung
Invers     = 112     'Attribut für Inversdarstellung
Intensiv   = 15      'Attribut für intensive Darstellung

'        * Windowvariablen *
Dim Menue$(2Ø)      'Anzahl der Pull-Down-Menüs
MaxChar = 4ØØØ: Dim Scr(MaxChar)   'Pufferspeicher (dynamisches Array)
MaxTab  = 1Ø:   Dim ScrTab(MaxTab) 'Window-Tabelle (dynamisches Array)
ScrPage    = 1       'Aktuelle Pufferseite
ScrTab(1)  = 1       'Ersten Tabelleneintrag initialisieren
WOffset    = 3       'Abstand Window-Rahmen zum Window-Inhalt
'-----------------------------------------------------------------------
```

Kümmern Sie sich bitte vorläufig überhaupt nicht um die »Window-Variablen«. Alle anderen Initialisierungen sind problemlos zu verstehen, die Tastendefinitionen, die Booleschen Variablenwerte und die Farbattribute. Auch der <Terminator$> in der Rubrik »Standardstrings« ist für uns zunächst uninteressant.

3.4 Die Module

Wir entwickeln nun eine kleine »Bibliothek« oder »Library«. Diese Library enthält einige nicht allzu schwierige, aber häufig benötigte Routinen. Die Library enthält sechs Prozeduren. Sie finden Sie auf der Diskette unter dem Namen LIBRARY.BAS. Das folgende Schema zeigt den Aufbau der Library.

```
'Prozedur 1
Sub Taste...            'Wartet auf eine Taste
...
End Sub...

...
...
...

'Prozedur 6
Sub LowUpCase...        'Klein- in Grossbuchstaben wandeln
...
End Sub...
```

Auf der Diskette befinden sich verschiedene Demoprogramme, die die Funktionsweise dieser Routinen demonstrieren. Alle Demoprogramme be-

ginnen mit drei Metabefehlen. Sehen wir uns als Beispiel das Demoprogramm FILL.BAS an. Dieses Demoprogramm ruft eine in der Library enthaltene Prozedur auf, die einen String mit Leerzeichen auf eine angegebene Länge auffüllt. Das Demoprogramm FILL.BAS:

```
'        * FILL – Demoprogramm *
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
REM $INCLUDE: 'library.bas'

Name$="Maier"
Print Name$ "enthält" Len(Name$) "Zeichen"
Call Fill(Name$,2Ø)
Print Name$ "enthält" Len(Name$) "Zeichen"
```

Sowohl in der Library als auch in den Demoprogrammen werden einige unserer globalen Variablen benötigt. Die beiden ersten Befehle sorgen dafür, daß die Variablen-Deklarationsdatei COMDEF.BAS und anschließend die Initialisierungs-Datei INIT.BAS kompiliert werden. Danach wird die Library mit den sechs Prozeduren kompiliert und erst zuletzt das eigentliche Demoprogramm. In jedem Demoprogramm werden der Reihe nach folgende Dateien kompiliert:

1. $INCLUDE: 'comdef.bas' bewirkt die Kompilierung der Datei COMDEF.BAS, in der die globalen Variablen deklariert werden.
2. $INCLUDE: 'init.bas' bewirkt die Kompilierung der Datei INIT.BAS, in der die globalen Variablen initialisiert werden.
3. $INCLUDE: 'library.bas' bewirkt die Kompilierung der Datei LIBRARY.BAS, in der sich die sechs Prozeduren befinden.
4. Nun erst wird das eigentliche Demoprogramm kompiliert.

Da alle Demoprogramme prinzipiell gleich aufgebaut sind, ist auch der Umgang mit ihnen immer gleich. Sie laden das betreffende Demoprogramm und starten es mit CTRL+R. Die drei $INCLUDE-Dateien werden automatisch mitkompiliert.

Achtung: $INCLUDE-Anweisungen setzen voraus, daß sich die zu »includierenden« Dateien im aktuellen Verzeichnis befinden!

COMPRESS und FILL

Die ganze Zeit erzähle ich von der Library und den zugehörigen Demoprogrammen, und wir haben noch keine Prozedur dieser Library entwickelt. Fangen wir mit den einfachsten Routinen an. FILL habe ich bereits erwähnt. Diese Prozedur füllt einen übergebenen String mit Leerzeichen auf. Das aufrufende Programm übergibt den aufzufüllenden String und die gewünschte Stringlänge.

```
'       Fill
'       ----
'       Funktion: Füllt String mit Spaces auf
'       Hin     : S$      : String
'                 SollLen : Soll-Länge des Strings

Sub Fill(S$, SollLen) Static
  S$ = S$ + Space$(SollLen - Len(S$))
End Sub
```

Das zugehörige Demoprogramm FILL.BAS:

```
'       * FILL - Demoprogramm *
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
REM $INCLUDE: 'library.bas'

Name$="Maier"
Print Name$ "enthält" Len(Name$) "Zeichen"
Call Fill(Name$,20)
Print Name$ "enthält" Len(Name$) "Zeichen"
```

Programmlauf:
```
Maierenthält 5 Zeichen
Maier        enthält 20 Zeichen
```

Beim Aufruf von FILL muß der String per Referenz übergeben werden. FILL soll den String verändern. Ob Sie die gewünschte Stringlänge nach Wert oder ebenfalls nach Referenz übergeben, ist Ihre Sache und interessiert die Prozedur überhaupt nicht.

COMPRESS ist die Umkehrung von FILL. COMPRESS entfernt alle Leerzeichen aus einem String, die dem letzten »Nicht-Leerzeichen« folgen. COMPRESS komprimiert den übergebenen String also auf seine »echte« Länge.

```
'       Compress
'       --------
'       Funktion: Komprimiert String auf echte Länge
'       Hin     : S$ : String
'       Zurück  : S$ : Komprimierter String

Sub Compress(S$) Static
  i = Len(S$)
  Komp: If i > 0 Then If Mid$(S$, i, 1) = " " Then_
          i = i - 1: Goto Komp
  S$ = Left$(S$, i)
End Sub
```

COMPRESS ist schon deutlich komplexer als FILL. Die Arbeitsweise: <i> ist ein <Zeiger> auf das gerade behandelte Zeichen im übergebenen String. <i> »zeigt« zuerst auf das letzte Zeichen des Strings (*i=Len(S$)*).

COMPRESS vergleicht das Zeichen, auf das <i> zeigt, mit einem Leerzeichen. Wenn es ein Leerzeichen ist, wird <i> um eins vermindert und zeigt

nun auf das vorhergehende Zeichen. Ist es kein Leerzeichen, wird die Schleife verlassen. Auf diese Weise »tastet« sich COMPRESS Zeichen für Zeichen vom Stringende nach vorne und zwar so lange, bis das erste Nicht-Leerzeichen erreicht ist. Dann wird <S$> der linke Stringteil bis zum ersten Leerzeichen zugewiesen.

Achtung: Beim »Zurücktasten« muß COMPRESS ständig prüfen, ob <i> noch größer als 0 ist (nur möglich, wenn der String ausschließlich Leerzeichen enthält). Den Wert 0 darf <i> niemals annehmen, da sonst *If Mid$(S$, i, 1)* zu einem Laufzeitfehler führt.

Wenn das Demoprogramm zur FILL-Routine um den Aufruf von COMPRESS erweitert wird, sollte aus »Maier « wieder »Maier« werden. Probieren Sie's aus. Das zugehörige Demoprogramm finden Sie unter dem Namen COMPRESS.BAS.

```
'       * COMPRESS - Demoprogramm *
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
REM $INCLUDE: 'library.bas'

Name$="Maier"
Print Name$ "enthält" Len(Name$) "Zeichen"
Call Fill(Name$,20)
Print Name$ "enthält" Len(Name$) "Zeichen"
Call Compress(Name$)
Print Name$ "enthält" Len(Name$) "Zeichen"
```

Programmlauf:
```
Maierenthält 5 Zeichen
Maier                enthält 20 Zeichen
Maierenthält 5 Zeichen
```

Sie sehen, es klappt. Der Aufruf von FILL füllt »Maier« mit Leerzeichen auf eine Länge von 20 Zeichen auf. Der Aufruf von COMPRESS entfernt alle überflüssigen Leerzeichen wieder.

LOWUPCASE und UPLOWCASE

Die Library enthält noch zwei »entgegengesetzte« Routinen. LOWUPCASE wandelt alle im übergebenen String enthaltenen Kleinbuchstaben in Großbuchstaben um (Low = niedrig / Up = hoch). UPLOWCASE wandelt genau umgekehrt Großbuchstaben in Kleinbuchstaben.

Das Prinzip der Wandlung beruht auf den ASCII-Codes der Zeichen. Schauen wir einen Auszug der ASCII-Tabelle an.

Großbuchstaben		Kleinbuchstaben	
Zeichen	Code	Zeichen	Code
A	65	a	97
B	66	b	98
.....		.....	
.....		.....	
Y	89	y	121
Z	90	z	122
.....			
.....			

Laut Tabelle unterscheiden sich die Codes eines Groß- vom entsprechenden Kleinbuchstaben immer um 32. Für die beiden Programme können wir also folgende Algorithmen verwenden:

1. UPLOWCASE: Jeder Großbuchstabe wird durch das Zeichen mit dem um 32 höheren Code ersetzt.
2. LOWUPCASE: Jeder Kleinbuchstabe wird durch das Zeichen mit dem um 32 niedrigeren Code ersetzt.

Die beiden Algorithmen sind nicht übel, aber einen Haken gibt es trotzdem: Die Umlaute passen nicht in dieses Schema. Sie sind ziemlich wahllos über die ASCII-Tabelle verstreut.

Ä	142	ä	132
Ö	153	ö	148
Ü	154	ü	129

Ich kann kein Schema in diesen Codes erkennen. Daher bleibt keine andere Wahl: Zur Wandlung der Umlaute benötigen wir drei separate Vergleiche. Kümmern wir uns zuerst um UPLOWCASE:

```
'       UpLowCase
'       ---------
'       Funktion: Wandelt alle im übergebenen String enthaltene
'                 Grossbuchstaben in Kleinbuchstaben
'       Hin     : S$
'       Zurück  : S$

Sub UpLowCase(S$) Static
  For i=1 to Len(S$)
    Char$ = Mid$(S$,i,1)
    If Char$ >= "A" And Char$ <= "Z" Then
      Char$ = Chr$(Asc(Char$) + 32)
    Else
      If Char$ = "Ä" Then Char$ = "ä"
      If Char$ = "Ö" Then Char$ = "ö"
      If Char$ = "Ü" Then Char$ = "ü"
    End If
    Mid$(S$, i ,1) = Char$
  Next i
End Sub
```

Mit einer FOR..NEXT-Schleife wird Zeichen für Zeichen des in <S$> übergebenen Strings behandelt. <i> ist wieder ein Zeiger auf das aktuelle Zeichen. Dieses aktuelle Zeichen wird <Char$> zugewiesen. Wenn <Char$> im Bereich der Großbuchstaben liegt (*If Char$ >= "A" And Char$ <= "Z"*), ermittelt die Prozedur mit der ASC-Funktion den ASCII-Code von <Char$>. Zu diesem Code wird 32 addiert, mit der Funktion CHR$ das zugehörige Zeichen erzeugt und im String gespeichert.

Wenn <Char$> kein »normaler« Großbuchstabe ist, kann es immer noch ein großgeschriebener Umlaut sein. Drei IF-Anweisungen vergleichen <Char$> mit den Umlauten »Ä«, »Ö« und »Ü«. Bei Gleichheit wird <Char$> in <S$> durch den zugehörigen Kleinbuchstaben ersetzt.

LOWUPCASE ist mit UPLOWCASE praktisch identisch. Der einzige Unterschied: Die Prozedur prüft diesmal, ob <Char$> im Bereich der Kleinbuchstaben liegt. Wenn ja, wird zum ASCII-Code nicht 32 addiert, sondern subtrahiert, um den zugehörigen Großbuchstaben zu erhalten.

```
'       LowUpCase
'       ---------
'       Funktion: Wandelt alle im übergebenen String enthaltene
'                 Kleinbuchstaben in Grossbuchstaben
'       Hin     : S$
'       Zurück  : S$
```

```
Sub LowUpCase(S$) Static
  For i=1 to Len(S$)
    Char$ = Mid$(S$,i,1)
    If Char$ >= "a" And Char$ <= "z" Then
      Char$ = Chr$(Asc(Char$) - 32)
    Else
      If Char$ = "ä" Then Char$ = "Ä"
      If Char$ = "ö" Then Char$ = "Ö"
      If Char$ = "ü" Then Char$ = "Ü"
    End If
    Mid$(S$, i ,1) = Char$
  Next i
End Sub
```

Beide Prozeduren können Sie mit dem Demoprogramm LOWUP.BAS erproben.

```
'       * LOWUPCASE/UPLOWCASE - Demoprogramm *
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
REM $INCLUDE: 'library.bas'

Name$="Peter Müller"
Print "Originalname:   " Name$
Call LowUpCase(Name$)
Print "Nach LowUpCase: " Name$
Call UpLowCase(Name$)
Print "Nach UpLowCase: " Name$
```

Programmlauf:
```
Originalname:   Peter Müller
Nach LowUpCase:.PETER MÜLLER
Nach UpLowCase: peter müller
```

Alle Zeichen des Originalnamens »Peter Müller« werden zuerst mit LOWUPCASE in Groß- und anschließend mit UPLOWCASE in Kleinbuchstaben gewandelt.

SEARCHOTHER

Sicher kennen Sie die INSTR-Funktion.

```
V=INSTR(X$,V$[,P])
```

INSTR gibt an, ab welcher Position der String <V$> in <X$> enthalten ist. Mit dem optionalen Parameter <P> können Sie bestimmen, ab welchem Zeichen <X$> durchsucht wird.

Leider vermisse ich oft eine Art »Umkehrung« von INSTR. In unseren Programmprojekten wird eine Funktion benötigt, die angibt, ab welcher Position in <X$> ein anderes (!) als ein in <V$> enthaltenes Zeichen auftritt!

Nehmen wir die Zeichenkette » Hermann Müller«. Angenommen, ich will wissen, ab welchem Zeichen der Vorname beginnt. Dann muß ich alle vorangehenden Leerzeichen überlesen. Ich brauche eine Routine, der ich mitteilen kann: »Suche mir die Position, ab der zum ersten Mal ein anderes Zeichen als ein Leerzeichen auftritt«.

Ein anderes Beispiel: Sie schreiben einen Editor, in dem der Benutzer mit CTRL+CURSOR RECHTS beziehungsweise CTRL+CURSOR LINKS wortweise (!) nach rechts und links springen kann. Nehmen wir folgenden »Programmtext«, eine einfache PRINT-Anweisung:

```
Print "Hallo" , "Test"
```

Gehen wir davon aus, daß sich der Cursor auf Zeichen Nummer 13 dieser Zeile befindet (auf den Anführungszeichen nach »Hallo«). Der Benutzer drückt CTRL+CURSOR RECHTS, um zum nächsten Wort zu springen, zu »Test«. Um zum nächsten Wort zu kommen, müssen Sie alle folgenden Trennzeichen überlesen, also die Zeichenfolge » , "«. Schön wäre eine Routine, der wir angeben: »Ermittle ab Zeichen Nummer 13 die erste Position im String, an der ein Nicht-Trennzeichen auftritt«. Genau diese Funktion erfüllt SEARCHOTHER. Der Routine werden drei Parameter übergeben:

- <S$>: Der zu durchsuchende String
- <Char$>: Ein String mit den zu überlesenden Zeichen
- <Ptr>: Position, ab der das Überlesen beginnen soll

<S$> und <Char$> werden nicht verändert, können also wahlweise nach Wert oder Referenz übergeben werden. <Ptr> müssen Sie nach Referenz übergeben. In <Ptr> übergibt SEARCHOTHER die ermittelte Position, ab der ein nicht in <Char$> enthaltenes Zeichen im String vorkommt. Wenn bis zum Stringende kein anderes Zeichen im String enthalten ist, übergibt die Prozedur den Wert null.

```
'       SearchOther
'       -----------
'       Funktion: Durchsucht 'S$' ab 'Ptr' und gibt an, ab
'                 welcher Position ein anderes Zeichen als
'                 eines der in 'Char$' enthaltenen auftritt.
'                 Enthält der String ab 'Ptr' kein anderes
'                 Zeichen, wird 'Ptr = Ø' übergeben.
'       Hin     : S$    : String
'                 Char$ : Zu überlesende Zeichen
'                 Ptr   : Suchstart
'       Zurück  : Ptr   : Gefundene Position
'                         (Ø = Suche war erfolglos)
```

```
Sub SearchOther(S$, Char$, Ptr) Static
  If Ptr <1 or Ptr > Len(S$) Then Ptr = Ø: Exit Sub
  While Instr(Char$, Mid$(S$, Ptr, 1)) <> Ø
    If Ptr = Len(S$) Then Ptr = False: Exit Sub
    Ptr = Ptr + 1
  Wend
End Sub
```

Bevor die Suche beginnt, prüft SEARCHOTHER, ob der Suchstart <Ptr> eine gültige Position im String definiert. <Ptr> darf nicht kleiner als eins und nicht größer als die Stringlänge sein. Wenn doch, wird <Ptr> der Wert null zugewiesen (=Suche war erfolglos) und die Prozedur mit EXIT SUB verlassen. Mit EXIT SUB kann eine Prozedur zu jedem Zeitpunkt verlassen werden. Diese Anweisung ist eine Art »Sprung zum Ende der Prozedur«.

Gibt <Ptr> eine gültige Zeichenposition an, beginnt die Suchschleife. Solange das untersuchte Zeichen in <Char$> enthalten ist, wird <Ptr> immer wieder um eins erhöht. Die Schleife tastet sich schrittweise zum Ende von <S$> vor. Sie wird verlassen, wenn das aktuelle Zeichen nicht in <Char$> enthalten ist. <Ptr> enthält nun die Position dieses Zeichens. In einem Sonderfall wird die Schleife vorzeitig mit EXIT SUB verlassen: Wenn <Ptr> mit der Stringlänge identisch ist, also gerade das letzte Zeichen untersucht wurde. <Ptr> wird vor dem Verlassen der Wert null zugewiesen, da der String ab dem angegebenen Suchstart offenbar nur Zeichen enthält, die in <Char$> enthalten sind.

Das Demoprogramm SOTHER.BAS zeigt Wirkungsweise und Anwendung der Prozedur.

```
'       * SEARCHOTHER – Demoprogramm *
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
REM $INCLUDE: 'library.bas'

Adresse$="Maier / Willi / Ludwigshafen / "
Char$=" /"
Ptr = 1 : Call SearchOther(Adresse$,Char$,Ptr)
Print Ptr
Ptr = 6 : Call SearchOther(Adresse$,Char$,Ptr)
Print Ptr
Ptr = 29: Call SearchOther(Adresse$,Char$,Ptr)
Print Ptr
```

Programmlauf:

```
1
9
Ø
```

Eine Adresse wird definiert, deren Einzelteile durch die Zeichenfolge » / « voneinander getrennt sind. <Char$> werden diese zu überlesenden »Trennzeichen« zugewiesen.

Beim ersten Aufruf wird ab Position 1 (Ptr=1) nach einem Nicht-Trennzeichen gesucht. Die Routine übergibt das Ergebnis 1, da dieses Zeichen bereits ein Nicht-Trennzeichen ist.

Der zweite Aufruf (Ptr=6) sucht Zeichen Nummer 6, also dem Leerzeichen nach »Maier«. Die Routine überliest die folgenden Trennzeichen »/ « und übergibt den Wert 9 (Position des Zeichens »W«).

Der dritte Aufruf zeigt, daß die Routine auch in »Grenzfällen« funktioniert. Gesucht wird ab Zeichen Nummer 29, dem Leerzeichen hinter »Ludwigshafen«. Die Routine übergibt eine 0, da nur noch Leerzeichen folgen.

TASTE

TASTE ist eine enorm wichtige Prozedur. Wie der Name sagt, wartet TASTE auf die Betätigung einer Taste. Bestimmt glauben Sie nun, ich will Sie auf den Arm nehmen. Um auf eine Taste zu warten, genügt schließlich eine Zeile:

```
Label: Key$ = Inkey$: If Key$ = "" Then Goto Label
```

Sicher, mit diesem Einzeiler wird auf eine Taste gewartet. Nur: Sie wissen nicht, ob es eine »normale« Taste war oder eine Sondertaste wie INS oder DEL mit einem Zwei-Zeichen-Code. Wenn Sie TASTE aufrufen, wissen Sie es!

Zur Erinnerung: Bei der Tastaturabfrage übergibt INKEY$ bei Sondertasten (alle Cursor- und Editiertasten) einen zwei Zeichen langen String. Das rechte Zeichen ist immer (!) CHR$(27), der Code der ESC-Taste und zeigt an, daß eine Sondertaste gedrückt wurde.

Das linke Zeichen unterscheidet die verschiedenen Sondertasten voneinander.

TASTE übergibt folgende Parameter an das aufrufende Programm:

- <Key$> enthält die gedrückte Taste. <Key$> enthält genau ein Zeichen. Wurde eine Sondertaste mit einem Zwei-Zeichen-Code gedrückt, trennt TASTE das linke Zeichen ab und übergibt nur das wichtigere rechte Zeichen.
- <Escape> ist ein Flag mit den beiden Zuständen <True> oder <False> (1 oder 0, erinnern Sie sich an die globalen Deklarationen). <True> bedeutet, daß eine Sondertaste gedrückt wurde, die einen Zwei-Zeichen-Code ergab. <False> zeigt eine normale alphanumerische Taste an.

<Key$> und <Escape> müssen per Referenz übergeben werden!

```
'       Auf Taste warten
'       ----------------
'       Funktion: Wartet auf Taste
'       Aufruf  : Call Taste(Key$, Escape)
'       Hin     :
'       Zurück  : Key$  : Taste
'                 Escape: Ø=Ein-Zeichen-Code/1=Zwei-Zeichen-Code

Sub Taste(Key$, Escape) Static
  Key$ = ""
  Escape = False
  While Key$ = ""
    Key$ = Inkey$
  Wend
  If Len(Key$) > 1 Then_
    Escape = True:_
    Key$ = Right$(Key$, 1)
End Sub
```

Die Arbeitsweise: <Key$> und <Escape> werden initialisiert. Die Routine wartet, bis eine Taste gedrückt wird. Enthält <Key$> nun mehr als ein Zeichen, wird das linke Zeichen entfernt und <Escape> der Wert <True> zugewiesen.

Das Demoprogramm TASTE.BAS:

```
'       * TASTE - Demoprogramm *
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
REM $INCLUDE: 'library.bas'

Print "Bitte 'a' drücken"
Call Taste(Key$, Escape)
Print Asc(Key$), Escape

Print "Bitte 'INS' drücken"
Call Taste(Key$, Escape)
Print Asc(Key$), Escape
```

Programmlauf:

```
Bitte 'a' drücken
97        Ø
Bitte 'INS' drücken
82        1
```

Das abgebildete Ergebnis des Programmlaufs tritt natürlich nur auf, wenn Sie den Anweisungen des Demoprogramms gehorsam folgen. »a« hat den ASCII-Code 97. <Escape> ist 0 (=False), da »a« keine Sondertaste ist. Entsprechend ist nach dem zweiten Aufruf und der Betätigung von INS <Escape> 1 (=True). Als Code der INS-Taste wird 82 übergeben.

Wenn Sie sich nun fragen, warum TASTE außer der Taste selbst das Flag <Escape> übergibt: Schauen Sie mal in die ASCII-Tabelle. 82 ist der Code von »R«. Die INS-Taste und das Zeichen »R« besitzen den gleichen Code. TASTE muß Sondertasten mit einem Flag kennzeichnen! Sonst kann das aufrufende Programm im Nachhinein nicht unterscheiden, ob die Sondertaste INS oder aber die alphanumerische Taste »R« gedrückt wurde.

So, nun kennen Sie den kompletten Inhalt von LIBRARY.BAS. Wenn Sie sich im Umgang mit Prozeduren und der Variablenübergabe nach Wert/Referenz noch unsicher fühlen, sollten Sie versuchen, die Library um Ihre eigenen Routinen zu erweitern.

3.5 Listing von LIBRARY.BAS

```
'       ******************************
'       *** LIBRARY: Mini-Routinen ***
'       ******************************


'       Auf Taste warten
'       ----------------
'       Funktion: Wartet auf Taste
'       Aufruf  : Call Taste(Key$, Escape)
'       Hin     :
'       Zurück  : Key$  : Taste
'                 Escape: Ø=Ein-Zeichen-Code/1=Zwei-Zeichen-Code

Sub Taste(Key$, Escape) Static
  Key$ = ""
  Escape = False
  While Key$ = ""
    Key$ = Inkey$
  Wend
  If Len(Key$) > 1 Then_
    Escape = True:_
    Key$ = Right$(Key$, 1)
End Sub
'-------------------------------------------------------------


'       SearchOther
'       -----------
'       Funktion: Durchsucht 'S$' ab 'Ptr' und gibt an, ab
'                 welcher Position ein anderes Zeichen als
'                 eines der in 'Char$' enthaltenen auftritt.
'                 Enthält der String ab 'Ptr' kein anderes
'                 Zeichen, wird 'Ptr = Ø' übergeben.
'       Hin     : S$    : String
'                 Char$ : Zu überlesende Zeichen
'                 Ptr   : Suchstart
```

```
'       Zurück  : Ptr    : Gefundene Position
'                          (Ø = Suche war erfolglos)

Sub SearchOther(S$, Char$, Ptr) Static
  If Ptr <1 or Ptr > Len(S$) Then Ptr = Ø: Exit Sub
  While Instr(Char$, Mid$(S$, Ptr, 1)) <> Ø
    If Ptr = Len(S$) Then Ptr = False: Exit Sub
    Ptr = Ptr + 1
  Wend
End Sub
'-------------------------------------------------------------

'       Compress
'       --------
'       Funktion: Komprimiert String auf echte Länge
'       Hin     : S$ : String
'       Zurück  : S$ : Komprimierter String

Sub Compress(S$) Static
  i = Len(S$)
  Komp: If i > Ø Then If Mid$(S$, i, 1) = " " Then_
          i = i - 1: Goto Komp
  S$ = Left$(S$, i)
End Sub
'-------------------------------------------------------------

'       Fill
'       ----
'       Funktion: Füllt String mit Spaces auf
'       Hin     : S$      : String
'                 SollLen : Soll-Länge des Strings

Sub Fill(S$, SollLen) Static
  S$ = S$ + Space$(SollLen - Len(S$))
End Sub
'-------------------------------------------------------------

'       UpLowCase
'       ---------
'       Funktion: Wandelt alle im übergebenen String enthaltene
'                 Grossbuchstaben in Kleinbuchstaben
'       Hin     : S$
'       Zurück  : S$

Sub UpLowCase(S$) Static
  For i=1 to Len(S$)
    Char$ = Mid$(S$,i,1)
    If Char$ >= "A" And Char$ <= "Z" Then
      Char$ = Chr$(Asc(Char$) + 32)
    Else
      If Char$ = "Ä" Then Char$ = "ä"
      If Char$ = "Ö" Then Char$ = "ö"
      If Char$ = "Ü" Then Char$ = "ü"
    End If
    Mid$(S$, i ,1) = Char$
  Next i
End Sub
'-------------------------------------------------------------
```

```
'       LowUpCase
'       ---------
'       Funktion: Wandelt alle im übergebenen String enthaltene
'                 Kleinbuchstaben in Grossbuchstaben
'       Hin     : S$
'       Zurück  : S$

Sub LowUpCase(S$) Static
  For i=1 to Len(S$)
    Char$ = Mid$(S$,i,1)
    If Char$ >= "a" And Char$ <= "z" Then
      Char$ = Chr$(Asc(Char$) - 32)
    Else
      If Char$ = "ä" Then Char$ = "Ä"
      If Char$ = "ö" Then Char$ = "Ö"
      If Char$ = "ü" Then Char$ = "Ü"
    End If
    Mid$(S$, i ,1) = Char$
  Next i
End Sub
'-----------------------------------------------------------
```

4

User-Libraries

Compiler-Sprachen wie QuickBASIC besitzen einen prinzipiellen Nachteil gegenüber Interpreter-Sprachen: Jede noch so winzige Änderung erfordert die Neukompilation des gesamten Programms. Außer dem Hauptprogramm werden auch alle $INCLUDE-Files »durchkompiliert«.

Sicher, im Moment ist dieser Nachteil nicht allzu schwerwiegend. Unsere Routinen-Sammlung LIBRARY.BAS enthält zwar einige recht nützliche Routinen, umfaßt aber insgesamt kaum mehr als 100 Zeilen Programmtext. Auf einem AT werden diese 100 Zeilen in weniger als einer Sekunde kompiliert und auch auf einem PC sind die Kompilierzeiten noch erträglich.

Nur, mit der Zeit wird LIBRARY.BAS umfangreicher. Sicher werden Sie unsere Library um mehrere eigene Routinen erweitern. Gehen wir davon aus, daß LIBRARY.BAS im Lauf der Zeit auf vielleicht 500 oder 1000 Programmzeilen anwächst.

Und nun beginnen Sie mit einem großen Programmprojekt, in das diese Library mit $INCLUDE eingebunden wird. Bei jeder Änderung des Hauptprogramms kompiliert QuickBASIC das gesamte Programm, also auch die 500–1000 Zeilen von LIBRARY.BAS. Glauben Sie mir, bei der Arbeit an einem umfangreichen Programm ist es ziemlich nervtötend, wenn jede noch so winzige Programmänderung wieder eine komplette Neukompilierung erforderlich macht.

Und das Schlimmste: In einem umfangreichen Programm werden Sie nicht nur LIBRARY.BAS benötigen. In einem professionellen Projekt wird zumindest noch eine vernünftige Eingaberoutine und eine Routine zur Verwaltung von Eingabemasken gebraucht. All diese umfangreichen »Module« werden ebenfalls mit $INCLUDE in das Hauptprogramm eingebunden. In der Praxis sieht das so aus:

```
'* Einbindung der benötigten Routinen *
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
REM $INCLUDE: 'library.bas'
REM $INCLUDE: 'eingabe.bas'
REM $INCLUDE: 'maske.bas'
```

```
'* Hauptprogramm *
...
```

Nehmen wir an, die eingebundenen Dateien umfassen insgesamt 2000 Programmzeilen, Ihr Hauptprogramm bisher nur 100 Zeilen. Bestimmt finden Sie es ärgerlich, wenn Sie im Hauptprogramm eine Zeile ändern und wieder alle 2000 Zeilen zu kompilieren sind, um die Änderung zu testen. Vor allem, da sich ja im größten Teil des Programms, in den eingebundenen $INCLUDE-Dateien, überhaupt nichts änderte!

Schön wäre es, wenn ein fertiges Modul wie eine Eingaberoutine nur einmalig kompiliert werden müßte und dann ständig zur Verfügung stünde. Genau das ermöglicht QuickBASIC mit den »User-Libraries«.

Ich kenne keinen anderen Compiler, der Ähnliches bietet. In dieser Beziehung ist QuickBASIC anderen Programmiersprachen um Längen voraus. Der sinnvolle Einsatz von User-Libraries räumt auf mit den immer wiederkehrenden zeitaufwendigen Kompilierläufen.

Das Prinzip:

1. User-Libraries bestehen aus einem oder mehreren kompilierten Programm(en), den »Modulen«. Beispiele für Module: eine Eingaberoutine, eine Routine zur Verwaltung von Pull-down-Menüs und so weiter.

2. Um ein Programm in eine User-Library »einzubinden«, muß es mit einer speziellen Option kompiliert werden. Das Kompilat – das erzeugte Maschinensprache-Programm – wird auf der Diskette in einer »Objektdatei« gespeichert.

3. Mit einem Hilfsprogramm werden beliebig viele dieser Objektdateien zu einer Datei zusammengefaßt, zur »User-Library«. Diese User-Library kann alle Unterprogramme und Prozeduren enthalten, die ein Programm zur Ausführung benötigt. Ein Modul der Library darf maximal 64 Kbyte groß sein. Der Umfang der gesamten User-Library wird jedoch nur durch den verfügbaren Speicherplatz Ihres Rechners eingeschränkt.

4. Wenn Sie am nächsten Tag programmieren wollen, teilen Sie QuickBASIC mit, daß Sie die User-Library verwenden wollen. QuickBASIC lädt die User-Library »resident« in den Speicher. Das heißt, die darin enthaltenen ausführbaren Programme befinden sich die ganze Zeit im Speicher, bis Sie QuickBASIC wieder verlassen.

5. Das Programm, das Sie entwickeln, kann unmittelbar auf die User-Library zugreifen. Das heißt, Sie können mit CALL NAME in der User-Library enthaltene Prozeduren aufrufen. Der Aufruf selbst unterscheidet sich nicht vom Aufruf einer Prozedur, die mit $INCLUDE eingebunden ist.

Der Unterschied: bei $INCLUDE wird der Programmtext (!) geladen und muß wie das Hauptprogramm kompiliert werden. Die residente User-Library enthält jedoch bereits ein lauffähiges Maschinensprache-Programm! Alle Routinen der User-Library können ohne Kompilierung vom Hauptprogramm genutzt werden. Nur das Hauptprogramm selbst wird bei jeder Änderung neu kompiliert.

Das klingt doch hervorragend, nicht wahr? Und tatsächlich kann ich mir das Erstellen größerer Programme ohne User-Libraries kaum noch vorstellen. Leider ist der Umgang mit User-Libraries das »dunkelste Kapitel« in der Dokumentation zu QuickBASIC. Das Kapitel »Die Erstellung von Benutzerbibliotheken« umfaßt drei Seiten! Mit genug Geduld finden Sie über das gesamte Handbuch verstreut noch weitere spärliche Hinweise. Der Erfolg dieser »Dokumentation«: Der QuickBASIC-Benutzer versucht sich am Aufbau einer eigenen User-Library, kämpft ständig mit immer neuen rätselhaften Fehlern und gibt irgendwann auf!

Ganz ehrlich: Auch ich selbst war zuerst am Verzweifeln. Ein halber Tag intensiven Experimentierens zusammen mit einem »Compiler-Spezialisten« war nötig, um alle Rätsel der User-Libraries aufzuklären.

4.1 Erstellung einer User-Library

Zur Erstellung einer User-Library benötigt QuickBASIC drei Dateien:

QB.EXE
USERLIB.EXE
BRUN20.LIB

Zuerst müssen Sie dafür sorgen, daß QuickBASIC auf diese Dateien auch tatsächlich zugreifen kann. Um das zu gewährleisten, gibt es verschiedene Möglichkeiten. Die einfachste: Nehmen Sie eine leere Diskette und kopieren Sie die erwähnten drei Dateien darauf. Diese Diskette ist nun Ihre »Arbeitsdiskette«, auf der Sie auch Ihre BASIC-Programme speichern. Die BASIC-Quelltexte und die zur Erstellung von User-Libraries benötigten Dateien befinden sich also gemeinsam auf einer Diskette. So gewährleisten Sie, daß QuickBASIC mit Sicherheit alle benötigten Dateien auch tatsächlich vorfindet.

Festplattenbesitzer können anstelle einer Diskette ein eigenes Unterverzeichnis QB zur Aufnahme dieser Dateien benutzen.

Ein Hinweis für Besitzer der Version 3.0: Die Datei BRUN20.LIB heißt in Ihrem QuickBASIC-System BRUN30.LIB!

Zum Experimentieren mit den User-Libraries werden wir mehrere BASIC-Programme benötigen, die sich auf der Diskette zum Buch im Verzeichnis DEMOS befinden. Kopieren Sie diese Programme bitte auf die Arbeitsdiskette. Es sind die Programme:

USER1.BAS
USER2.BAS
GLOBAL.BAS
GDEMO.BAS

Auf der Arbeitsdiskette (beziehungsweise im Festplatten-Subdirectory QB) befinden sich nun insgesamt sieben Dateien: die vier BASIC-Programme und die Dateien QB.EXE, USERLIB.EXE, BRUN20.LIB (BRUN30.LIB bei der QuickBASIC-Version 3.0). Um ein Modul in die User-Library einzubinden, sind folgende Schritte nötig:

1. Zuerst wird das Modul mit der Option »Obj (BRUN.LIB)« kompiliert. Das erzeugte Kompilat wird von QuickBASIC auf Diskette/Festplatte in einer »Objektdatei« gespeichert. Eine Objektdatei erkennen Sie am Namenszusatz .OBJ. Angenommen, Ihr BASIC-Quelltext heißt TEST.BAS. Nach der Kompilierung befindet sich das Kompilat auf der Diskette unter dem Namen TEST.OBJ.

2. Mit dem Programm BUILDLIB wird nun aus einer oder mehreren Objektdatei(en) eine User-Library erstellt. Zuvor müssen Sie QuickBASIC verlassen und auf die Betriebssystem-Ebene zurückkehren. Beim Aufruf von BUILDLIB geben Sie alle Objektdateien an, die in die User-Library einzubinden sind und trennen die einzelnen Dateinamen durch Leerzeichen voneinander. Der Aufruf endet mit einem Semikolon.

   ```
   BUILDLIB Datei1 Datei2 ... DateiN;
   ```

3. BUILDLIB erzeugt die User-Library USERLIB.OBJ, die nun alle angegebenen Objektdateien enthält (nicht den Quelltext, sondern das Kompilat).

   Beispiel: `BUILDLIB EINGABE MASKE ROUTINEN;`

   erstellt eine User-Library, die die drei Objektdateien EINGABE.OBJ, MASKE.OBJ und ROUTINEN.OBJ enthält.

4. Um die User-Library zu benutzen, rufen Sie QuickBASIC mit dem Zusatz /L auf.

   ```
   QB/L
   ```

Beim Aufruf lädt QuickBASIC die User-Library resident in den Speicher.

Das Ganze nochmal in einer Kurzfassung:

1. BASIC-Programm mit der Option »Obj (BRUN.LIB)« kompilieren
2. QuickBASIC verlassen
3. Mit »BUILDLIB Datei1 Datei2 ... DateiN;« die User-Library erzeugen
4. QuickBASIC und die User-Library mit »QB/L« laden

Zuerst binden wir das Programm USER1.BAS ein. Es enthält nur eine Prozedur, USERDEMO1.

```
'User-Libraries: Demo-Prozedur
Sub UserDemo1 Static
  Input "Geben Sie eine Zahl ein"; Zahl1
  Input "Geben Sie eine zweite Zahl ein"; Zahl2
  Print "Die Summe beider Zahlen ist" Zahl1 + Zahl2
End Sub
```

Ziemlich sinnlos, nicht wahr? USERDEMO1 fragt nach zwei Zahlen, addiert sie und gibt das Resultat auf dem Bildschirm aus. Gehen Sie nun bitte so vor:

1. Laden Sie USER1.BAS unter QuickBASIC von der Arbeitsdiskette (Platte: Verzeichnis QB).
2. Kompilieren Sie die Prozedur mit der Option »Obj (BRUN.LIB)«. Das Kompilat wird von QuickBASIC auf der Diskette unter dem Namen USER1.OBJ gespeichert.
3. Verlassen Sie QuickBASIC und kehren Sie auf die Betriebssystem-Ebene zurück.
4. Geben Sie ein: `BUILDLIB USER1;`
   Das Programm BUILDLIB erzeugt eine User-Library mit dem Namen USERLIB.EXE, die unsere kleine Prozedur in ausführbarer Form enthält.
5. Rufen Sie wieder QuickBASIC auf, diesmal jedoch mit QB/L, um die User-Library resident zu laden.

Fassen wir zusammen: Wir erzeugten eine User-Library, die die Prozedur USERDEMO1 enthält. Die User-Library wurde beim erneuten Aufruf von QuickBASIC resident geladen und kann nun benutzt werden. Zum Testen genügt ein aufrufendes Programm, das aus einer einzigen Zeile besteht:

```
Call UserDemo1
```

Dieses »Hauptprogramm« starten Sie wie gewohnt mit CTRL+R. Wenn alles geklappt hat, wird die USERDEMO1 aufgerufen. Sie werden nach zwei Zahlen gefragt; die Zahlen werden addiert und die Summe ausgegeben.

Der Witz dabei: Beim Kompilieren mit CTRL+R wird nur das Hauptprogramm kompiliert, nicht die Prozedur!

Ändern Sie das Hauptprogramm nun bitte in:

```
For i=1 to 3
  Call UserDemo1
Next i
```

Wieder wird nur das drei Zeilen lange Hauptprogramm kompiliert. Stellen Sie sich vor, USERDEMO1 bestünde aus 1000 oder 2000 Programmzeilen. Es ist einfach fantastisch, daß dieser riesige Programmtext nach der Einbindung in eine User-Library nie wieder mitkompiliert würde!

Wie gesagt, in eine User-Library kann nicht nur ein Programm eingebunden werden. Die Anzahl der eingebundenen Programme ist prinzipiell nur durch die Größe des Hauptspeichers beschränkt. Wir werden nun zusätzlich noch die in USER2.BAS enthaltene Prozedur einbinden.

```
'User-Libraries: Demo-Prozedur Nummer 2
Sub UserDemo2(N) Static
  For i=1 to N
    Print i;
  Next i
End Sub
```

USERDEMO2 gibt einfach die Zahlen 1–N auf dem Bildschirm aus. <N> wird vom aufrufenden Programm übergeben. Laden Sie bitte USER2.BAS und kompilieren Sie wieder mit der Option »Obj (BRUN.LIB)«. Das erzeugte Kompilat speichert QuickBASIC unter dem Namen USER2.OBJ.

Nun soll eine User-Library erstellt werden, die aus den beiden Dateien USER1.OBJ und USER2.OBJ besteht. Beide Dateien geben Sie beim Aufruf von BUILDLIB an. Verlassen Sie QuickBASIC und geben Sie ein:

```
BUILDLIB USER1 USER2;
```

Wieder erstellt BUILDLIB eine User-Library mit dem Namen USERLIB.EXE, in der nun beide Programme enthalten sind. Rufen Sie QuickBASIC auf mit:

```
QB/L
```

um die neue User-Library resident zu laden. Testen Sie die User-Library mit folgendem Hauptprogramm:

```
Call UserDemo1
N=10: Call UserDemo2((N))
```

Nach CTRL+R wird nur das aus zwei Zeilen bestehende Hauptprogramm kompiliert. Die Prozeduren USERDEMO1 (Zahlenaddition) und USERDEMO2 (Ausgabe der Zahlen 1–N) werden nacheinander aufgerufen.

Ebenso wie in diesem Versuch müssen Sie BUILDLIB immer alle (!) einzubindenden Objektdateien angeben. Es ist nicht möglich, an eine bereits existierende User-Library eine Objektdatei »anzuhängen«.

Leider kann ich Ihnen nicht »über die Schulter schauen«. Ich hoffe, daß alles geklappt hat. Im Moment kommt Ihnen die Erstellung einer User-Library bestimmt ziemlich umständlich vor. Unsere beiden winzigen Prozeduren sind einfacher als $INCLUDE-Files einzubinden und blitzschnell mitkompiliert.

Bedenken Sie jedoch, daß die Erstellung einer neuen User-Library in der Praxis relativ selten vorkommt. Wenn Sie sich später mit User-Libraries auskennen, haben Sie eine »Standard-Library«, die Sie für all Ihre Programme verwenden. Diese Library enthält alle immer wieder benötigten Programme. Sie wird beim Aufruf von QuickBASIC resident geladen und steht ohne Kompilierung jederzeit zur Verfügung.

Erst nach längerer Zeit, wenn sich eine Anzahl neuer vielseitiger Routinen angesammelt hat, binden Sie diese ebenfalls in die User-Library ein.

Arbeitsaufwendig sind User-Libraries nur bei falscher Anwendung. Wenn Sie ökonomisch vorgehen wollen, sollten Sie vermeiden:

- Ständig wegen jeder noch so winzigen neuen Prozedur, die man »eventuell einmal gebrauchen könnte«, die User-Library neu zu erstellen. Binden Sie neue Routinen wie gehabt mit $INCLUDE ein. Wenn Ihnen das ständige Durchkompilieren der $INCLUDE-Dateien auf die Nerven geht, dann ist der Zeitpunkt zur Erweiterung der User-Library gekommen.
- Auf gar keinen Fall sollten Sie eine neue Prozedur sofort (!) in die User-Library einbinden. Seien wir ehrlich: Frisch erstellte Programme sind nie fehlerfrei. Machen Sie nicht meinen Fehler: Aus lauter Begeisterung über die User-Libraries habe ich natürlich jede frisch entwickelte Routine sofort in meine User-Library eingebunden. Nur um kurz darauf festzustellen, daß die Routine noch einen Fehler enthält. Das Resultat: Fehler verbessern, neu kompilieren, User-Library neu erstellen. Natürlich entdeckte ich in der verbesserten Version kurz darauf wieder einen Fehler, und das ganze Spielchen wiederholte sich! Also bitte: Warten Sie mit dem Einbinden, bis sich eine Routine in der täglichen Arbeit als hundertprozentig (???) fehlerfrei erwies.

Übrigens: USERLIB ist der »voreingestellte« Standardname für eine User-Library. Wenn Sie wollen, können Sie auch andere Namen wählen, die dann jedoch explizit anzugeben sind.

```
BUILDLIB DEMO1 DEMO1,TESTLIB;
```

Der Name der User-Library wird durch ein Komma von den Objektdateien getrennt am Ende des Aufrufs angegeben. BUILDLIB gibt der erzeugten User-Library den Namen TESTLIB. Um TESTLIB resident zu laden, müssen Sie den vollständigen (!) Namen auch beim Aufruf von QuickBASIC angeben.

```
QB/L TESTLIB.EXE
```

Diese Möglichkeit erwähne ich nur der Vollständigkeit wegen. Im folgenden werden wir ausschließlich mit der Voreinstellung arbeiten. Jede von uns erstellte User-Library heißt USERLIB.

4.2 Ein Standardfehler

Sehr häufig ist folgender Fall: Sie erstellen eine User-Library mit dem Programm BUILDLIB. Danach laden Sie QuickBASIC und die User-Library mit *QB/L* und probieren aus, ob auch alles klappt wie gewünscht.

Nun stellen Sie fest, daß eine Prozedur der User-Library nicht so arbeitet, wie sie sollte. Das heißt also: Programm ändern, in dem sich die Prozedur befindet, wieder mit »Obj (BRUN.LIB)« kompilieren und erneut mit BUILDLIB die geänderte User-Library erstellen.

Sie laden also den zugehörigen Quelltext, ändern ihn und kompilieren das Programm erneut. Und nun stellen Sie mit Erstaunen fest, daß sich QuickBASIC weigert, das Programm zu kompilieren. Statt dessen erscheint die Fehlermeldung DOPPELTE DEFINITION IN BENUTZER-BIBLIOTHEK.

Spielen wir das Ganze an einem Beispiel durch. Zuletzt erstellten Sie eine User-Library, die die kompilierten Programme USER1.OBJ und USER2.OBJ enthielt. Sie riefen QuickBASIC mit *QB/L* auf, die User-Library ist also resident im Speicher vorhanden.

Nehmen wir nun an, die Prozedur USERDEMO1 soll nicht zwei, sondern drei Zahlen addieren. Wir müssen den zugehörigen Quelltext USER1.BAS ändern. Laden Sie bitte USER1.BAS.

```
'User-Libraries: Demo-Prozedur
Sub UserDemo1 Static
  Input "Geben Sie eine Zahl ein"; Zahl1
  Input "Geben Sie eine zweite Zahl ein"; Zahl2
  Print "Die Summe beider Zahlen ist" Zahl1 + Zahl2
End Sub
```

Ändern Sie den PRINT-Befehl und fügen Sie einen weiteren INPUT-Befehl ein.

```
'User-Libraries: Demo-Prozedur
Sub UserDemo1 Static
  Input "Geben Sie eine Zahl ein"; Zahl1
  Input "Geben Sie eine zweite Zahl ein"; Zahl2
  Input "Geben Sie eine dritte Zahl ein"; Zahl3
  Print "Die Summe der drei Zahlen ist" Zahl1 + Zahl2 + Zahl3
End Sub
```

Kompilieren Sie das Programm wie gewohnt mit der Option »Obj (BRUN.LIB)«. Und nun erhalten Sie die erwähnte Fehlermeldung DOPPELTE DEFINITION IN BENUTZER-BIBLIOTHEK. Der Cursor – der den genauen Fehlerort spezifiziert – befindet sich in der Zeile mit der SUB-Anweisung. Offenbar ist QuickBASIC mit der Deklaration der Prozedur USERDEMO1 nicht einverstanden.

Warum? Weil bereits in der residenten User-Library eine Prozedur mit dem Namen USERDEMO1 enthalten ist. Das Gesamtprogramm besteht für QuickBASIC aus dem Inhalt der geladenen User-Library und Ihrem Quelltext. Beim Kompilieren des Quelltextes stellt QuickBASIC fest, daß eine Prozedur namens USERDEMO1 erzeugt werden soll, obwohl bereits eine Prozedur mit diesem Namen im Gesamtprogramm vorhanden ist (eben die in der User-Library (!) enthaltene Prozedur USERDEMO1).

Das Problem ist, daß dieser Fehler nicht leicht auffällt, da die User-Library für uns nicht sichtbar ist. Im folgenden Programm erkennen Sie den Fehler sofort.

```
Sub Demo Static
  Print "Hallo"
End Sub

Sub Demo Static
  Print "Guten Tag"
End Sub
```

In diesem Programm würde Ihnen die doppelte Deklaration der Prozedur DEMO sofort auffallen. Eine resident geladene User-Library besitzt nun mal die Eigenschaft, nicht im Quelltext, sondern gewissermaßen unsichtbar im Hintergrund vorhanden zu sein. Daher ist es leicht, zu vergessen, daß es in der User-Library bereits eine Prozedur mit dem Namen USERDEMO1 gibt.

Daraus folgt, daß es ziemlich umständlich ist, die Prozedur USERDEMO1 zu ändern. Sie müssen QuickBASIC verlassen und ohne den Parameter »/L« erneut aufrufen. Die User-Library mit der Prozedur USERDEMO1 wird nicht (!) mitgeladen.

Sie können den Quelltext der Prozedur laden, ändern und ohne Probleme kompilieren. Anschließend verlassen Sie QuickBASIC, benutzen die erzeugte Datei USER1.OBJ zum Aufbau der neuen User-Library und laden diese resident mit *QB/L*. Sollten Sie jedoch feststellen, daß die Prozedur immer noch nicht Ihren Wünschen entspricht, wiederholt sich das Spielchen.

Zusammenfassung:

- QuickBASIC betrachtet den gerade editierten Programmtext und eine resident geladene User-Library als eine Einheit, ein Gesamtprogramm, in dem selbstverständlich jeder Prozedurname nur einmalig vorkommen darf.
- Daher meldet QuickBASIC DOPPELTE DEFINITION IN BENUTZER-BIBLIOTHEK beim Kompilieren, wenn im editierten Programmtext ein Prozedurname verwendet wird, den bereits die residente User-Library enthält.
- Daraus folgt zwingend eine leider recht umständliche Vorgehensweise beim Ändern eines »Moduls« der User-Library: Sie müssen QuickBASIC verlassen und ohne die User-Library laden. Nun können Sie den zugehörigen Programmtext laden, editieren, kompilieren und die geänderte User-Library erstellen.

4.3 Globale Variablen in User-Libraries

Bei der Verwendung von User-Libraries gibt es eine »Kleinigkeit«, die einen zur Verzweiflung bringen kann. Bisher konnten wir Prozeduren völlig unverändert in eine User-Library einbinden. Leider gilt das nur, solange keine globalen Variablen verwendet werden. Ich wette mit Ihnen: In dem Moment, in dem Sie ein Programm in die User-Library einbinden, das globale Variablen verwendet, klappt nichts mehr! Warum, demonstriere ich an dem Programm GLOBAL.BAS, das sich ebenfalls im Verzeichnis DEMOS befindet.

```
'Hauptprogramm
Common Shared N          'N : globale Variable
N = 1Ø
Call GlobalDemo

'Prozedur
```

```
Sub GlobalDemo Static
  For i=1 to N            'Benutzung der globalen Var. N
    Print "Hallo"
  Next i
End Sub
```

GLOBAL.BAS enthält die Prozedur GLOBALDEMO und ein aufrufendes Hauptprogramm. Am Programmanfang wird »N« als globale Variable deklariert. Das Hauptprogramm weist »N« den Wert zehn zu und ruft die Prozedur auf. GLOBALDEMO gibt N-mal »Hallo« aus, also zehnmal. Da »N« als globale Variable deklariert wurde, kann jeder Programmteil auf »N« zugreifen, also auch die Prozedur GLOBALDEMO.

Theoretisch könnten wir die Prozedur vom aufrufenden Programm trennen und in eine User-Library einbinden.

```
'Prozedur
Sub GlobalDemo Static
  For i=1 to N            'Benutzung der globalen Var. N
    Print "Hallo"
  Next i
End Sub
```

Nur, woher soll QuickBASIC beim Kompilieren dieses Programms wissen, daß »N« eine globale Variable ist? Sie kompilieren ein eigenständiges Programm! In diesem Programm kommt keine COMMON SHARED-Anweisung vor. Für dieses (!) Programm ist »N« keine globale Variable. QuickBASIC »weiß« beim Kompilieren nicht, daß »N« global sein soll.

Denken Sie immer daran: Eine User-Library besteht aus einzeln kompilierten Programmen! Wenn eine bestimmte Variable in all diesen Programmen bekannt (global) sein soll, muß jedes einzelne Programm eine entsprechende COMMON SHARED-Anweisung enthalten! Auf die Arbeitsdiskette kopierten Sie zu Beginn unserer Versuche auch die Datei GDEMO.BAS. Dieses Programm enthält die korrigierte Version der Prozedur GLOBALDEMO.

```
Common Shared N          'N : globale Variable

'Prozedur
Sub GlobalDemo Static
  For i=1 to N            'Benutzung der globalen Var. N
    Print "Hallo"
  Next i
End Sub
```

In diesem Programm wird »N« zu Beginn als globale Variable deklariert. Laden Sie bitte GDEMO.BAS und kompilieren Sie mit »Obj (BRUN.LIB)«. Verlassen Sie QuickBASIC und erzeugen Sie eine User-Library, die das kompilierte Programm enthält:

```
BUILDLIB GDEMO;
```

Rufen Sie QuickBASIC und die User-Library auf und geben Sie bitte folgendes Programm ein:

```
'Hauptprogramm
Common Shared N          'N : globale Variable
N = 1Ø
Call GlobalDemo
```

Nach dem Befehl zur Ausführung wird wie gewünscht zehnmal »Hallo« ausgegeben. »N« ist nicht nur für das aufrufende Hauptprogramm global, sondern auch für das in der User-Library enthaltene Programm GDEMO.

Dieses Problem zu verstehen, ist so wichtig, da kaum ein größeres Programm ohne globale Variablen auskommt. Daher müssen Sie sich darüber im Klaren sein: Kompilieren bedeutet, daß Sie ein eigenständiges »Programm-Modul« erzeugen! In diesem Modul enthaltene Prozeduren können zwar von anderen Programmen aus aufgerufen werden. Jedes Programm verwendet jedoch wie immer seine eigenen Variablen!

Die Anweisung COMMON SHARED sagt nur aus, daß die angegebenen Variablen in diesem einen Programm bekannt sein sollen. Für alle anderen Programme sind diese Variablen unbekannt. Soll eine Variable in mehreren Programmen bekannt sein, muß sie in jedem einzelnen Programm mit COMMON SHARED als global deklariert werden!

Zusammenfassung

1. Kompilieren heißt, ein eigenständiges Programm erzeugen. Die Erzeugung einer User-Library verbindet mehrere unabhängige Programme. Jede in einem Programm enthaltene Prozedur kann nun von einem anderen Programm aus aufgerufen werden. Wie immer benutzt jedoch jedes Programm seine eigenen Variablen und kennt nicht die Variablen eines anderen Programms.

2. COMMON SHARED deklariert Variablen als global nur für jenes Programm, in dem sich die Anweisung befindet. Für alle anderen Programme sind die angegebenen Variablen unbekannt.

3. Soll eine Variable in mehreren Programmen bekannt sein, muß sie für jedes einzelne dieser Programme mit COMMON SHARED als global deklariert werden!

4.4 Einbindung vorhandener Module

Achtung: Die Programme, die wir nun entwickeln, finden Sie nicht mehr im Verzeichnis DEMOS, sondern in MODULE! DEMOS enthält die zugehörigen Demoprogramme, die die Funktion unserer Programme testen.

Sie erinnern sich an unsere kleine Routinen-Sammlung? LIBRARY.BAS enthält sechs immer wieder benötigte Prozeduren. Da dieses Programm immer wieder benötigt wird, ist eine Einbindung in die User-Library sinnvoll. Wie wir sahen, sind jedoch zuvor Änderungen erforderlich, wenn globale Variablen benutzt werden.

Globale Variablen werden nur in einer Prozedur von LIBRARY.BAS benötigt: in TASTE. TASTE verwendet die globalen Variablen <True> und <False>. Es würde ausreichen, eine COMMON SHARED-Anweisung in LIBRARY.BAS einzufügen, um diese beiden Variablen als global zu deklarieren.

Erinnern Sie sich auch an COMDEF.BAS?

```
'       ******************************************************
'       *   COMDEF.BAS = Common Definitions meiner Programme  *
'       ******************************************************

DefInt a-z      'Grundtyp numerischer Variablen: INTEGER
Option Base 1   'Arrayuntergrenze: 1

'       ************************************
'       *   Globale Variablen/Konstanten   *
'       ************************************

...
...
...
```

Diese Datei enthält fast nur COMMON SHARED-Anweisungen, mit denen jene Variablen als global deklariert werden, die wir in unseren Programmprojekten häufig benötigen werden.

In der Routinen-Sammlung wird momentan von diesen globalen Variablen nur <True> und <False> benötigt. Möglicherweise erweitern Sie die Library mit der Zeit um einige Routinen. Dann ist es nicht unwahrscheinlich, daß Sie zum Beispiel die globalen »Farbvariablen« (<CharCol>, <BackCol>, ...) benötigen oder einige der »Tastenvariablen« (<zEsc$>, <zIns$>, <zDel$>, ...).

Es ist daher sinnvoll, in allen Programmen, die in eine User-Library eingebunden werden, »Nägel mit Köpfen« zu machen und für jedes einzelne Programm alle Standard-Variablen als global zu deklarieren. Diese Standard-

Variablen enthält unsere Deklarationsdatei COMDEF.BAS. Wir fügen alle Deklarationen von COMDEF.BAS am Anfang von LIBRARY.BAS ein.

Entweder tippen Sie den kompletten Inhalt von COMDEF.BAS ab oder Sie vereinfachen sich die Arbeit: Mit einer $INCLUDE-Anweisung am Anfang der Routinen-Sammlung wird die »Deklarationsdatei« COMDEF.BAS eingebunden.

Auf der Diskette zum Buch befindet sich die Datei ROUTINEN.BAS. ROUTINEN.BAS finden Sie nicht im Verzeichnis DEMOS, sondern in MODULE. Kopieren Sie diese Datei bitte auf die Arbeitsdiskette.

ROUTINEN.BAS ist bis auf die zusätzliche $INCLUDE-Anweisung mit unserer Routinen-Sammlung LIBRARY.BAS identisch.

ROUTINEN.BAS:

```
'       ******************************
'       *** LIBRARY: Mini-Routinen ***
'       ******************************

REM $INCLUDE: 'comdef.bas'
'-------------------------------------------------------------
'       Auf Taste warten
'       ----------------
'       Funktion: Wartet auf Taste
'       Aufruf  : Call Taste(Key$, Escape)
'       Hin     :
'       Zurück  : Key$  : Taste
'                 Escape: Ø=Ein-Zeichen-Code/1=Zwei-Zeichen-Code

Sub Taste(Key$, Escape) Static
  Key$ = ""
  Escape = False
  While Key$ = ""
    Key$ = Inkey$
  Wend
  If Len(Key$) > 1 Then_
    Escape = True:_
    Key$ = Right$(Key$, 1)
End Sub
'-------------------------------------------------------------
...
...
...
'-------------------------------------------------------------

Sub LowUpCase(S$) Static
  For i=1 to Len(S$)
    Char$ = Mid$(S$,i,1)
    If Char$ >= "a" And Char$ <= "z" Then
      Char$ = Chr$(Asc(Char$) - 32)
    Else
```

```
      If Char$ = "ä" Then Char$ = "Ä"
      If Char$ = "ö" Then Char$ = "Ö"
      If Char$ = "ü" Then Char$ = "Ü"
    End If
    Mid$(S$, i ,1) = Char$
  Next i
End Sub
'-----------------------------------------------------------
```

Um nicht unnötig Seiten zu verschwenden, ist nur ein Teil des kompletten Listings abgebildet: der Anfang von ROUTINEN.BAS mit der $INCLUDE-Anweisung zur Einbindung der Datei COMDEF.BAS, die erste Prozedur TASTE und die letzte Prozedur LOWUPCASE.

Kompilieren Sie ROUTINEN.BAS nun bitte wieder mit der Option »Obj (BRUN.LIB)« und erzeugen Sie mit

```
BUILDLIB ROUTINEN;
```

eine User-Library, die nur dieses eine Programm enthält. Die vorhergehenden User-Libraries dienten nur Demonstrationszwecken und enthielten ziemlich unsinnige Programme. Diese User-Library ist zum ersten Mal wirklich sinnvoll.

Sie können nun beliebige Programme erstellen, die unsere sechs kleinen Prozeduren verwenden, ohne daß diese immer wieder mitkompiliert werden (Voraussetzung: Sie rufen QuickBASIC mit *QB/L* auf, um die User-Library resident zu laden).

Wir werden diese User-Library in den folgenden Kapiteln erweitern und weitere Programme einbinden. Das Programm ROUTINEN wird jedoch immer in unserer User-Library enthalten sein.

Ich empfehle Ihnen dringend, zur Deklaration globaler Variablen immer eine eigene Deklarationsdatei (hier: COMDEF.BAS) zu verwenden. Diese Deklarationsdatei wird wie im folgenden Schema mit $INCLUDE in jedes Programm eingebunden, das in die User-Library kommen soll.

```
--------------------------
'Programm1: ROUTINEN
REM $INCLUDE: 'comdef.bas'
...
...
--------------------------

--------------------------
'Programm2: EINGABE
REM $INCLUDE: 'comdef.bas'
...
...
--------------------------
```

```
-------------------------
'Programm3: MASKE
REM $INCLUDE: 'comdef.bas'
...
...
-------------------------
```

Dieses Schema soll eine User-Library darstellen, die die drei Programme ROUTINEN (Routinen-Sammlung), EINGABE (Eingaberoutine) und MASKE (Maskensteuerung) enthält. Wenn alle drei Programme nicht nur aus einer einzigen Prozedur bestehen, sondern recht umfangreich sind, verwendet jedes Programm ziemlich sicher einige unserer Standard-Variablen wie zum Beispiel <True>, <False>, <CharCol> oder <zReturn$>.

Da in jedes Programm die Deklarationsdatei COMDEF.BAS eingebunden wird, kennt jedes Programm alle verwendeten globalen Variablen. Der Vorteil der Deklarationsdatei: Wenn Sie einige zusätzliche globale Variablen in den einzelnen Programmen benötigen, müssen Sie nicht jedes einzelne Programm um die entsprechenden COMMON SHARED-Anweisungen erweitern!

Angenommen, Sie benötigen in einigen Programmen zusätzlich die globale Variable <Zahl>. Ohne per $INCLUDE eingebundene Deklarationsdatei müssen Sie jedes Programm um die Anweisung

```
COMMON SHARED Zahl
```

ergänzen. Die Gefahr dabei: Es ist sehr leicht, in einem der Programme diese zusätzliche COMMON SHARED-Anweisung zu vergessen. Und schon ist <Zahl> für das betreffende Programm nicht global!

Bei Verwendung der Deklarationsdatei genügt es, COMDEF.BAS um die die COMMON SHARED-Anweisung zu erweitern. Sie müssen zwar ebenfalls alle drei Programme neu kompilieren. Dank der in jedem Programm enthaltenen Anweisung

```
$INCLUDE: 'comdef.bas'
```

wird jedoch die erweiterte Deklarationsdatei automatisch mitkompiliert. Sie ersparen sich nicht nur Tipparbeit. Es ist gar nicht möglich, in einem der drei Programme die COMMON SHARED-Anweisung zu vergessen und dadurch unnötige Fehler zu erzeugen.

4.5 Das getrennte Kompilierverfahren

Um ein BASIC-Programm zu kompilieren und ein Objektmodul zu erzeugen, müssen Sie nicht unbedingt mit QuickBASIC »in Dialog treten«. Das »getrennte Kompilierverfahren« erlaubt Ihnen, in der Betriebssystem-Ebene zu bleiben. Wenn Sie den Aufruf von QuickBASIC mit einem Semikolon abschließen, wird das angegebene Programm kompiliert, ohne daß Sie in den Editier-Modus gelangen.

```
QB USER1;

Microsoft QuickBASIC Compiler
Version 2.Ø1

(C) Copyright Microsoft Corp. 1982, 1983, 1984, 1985, 1986, 1987

1ØØ Zeilen kompiliert
Ø Schwerwiegende(r) Fehler
```

Der Aufruf »QB USER1;« bewirkt, daß QuickBASIC das BASIC-Programm USER1 kompiliert und eine Objektdatei erzeugt. QuickBASIC meldet Ihnen, wieviele Zeilen kompiliert wurden und wie viele Fehler beim Kompilieren auftraten. Anschließend landen Sie wieder im Betriebssystem, das auf die nächste Anweisung wartet.

Unter den verschiedenen Optionen wählen Sie durch Angabe von Parametern, die durch den Schrägstrich (»/«) voneinander getrennt werden. Allgemein lautet der Aufruf:

```
QB Dateiname [Optionsliste];
```

Jeder Parameter in der Optionsliste entspricht einer der Optionen des »KOMPILIEREN...«-Menüs.

Parameter	Option
Kompilierungsoptionen	
/d	Austesten
/r	Datenfelder in Zeilenanordnung
/s	Zeichenfolgedaten minimieren
/e	On Error
/x	Resume Next
/w	Ereigniserfassung
/v	Prüfen zwischen Anweisungen

Parameter	Option
Ausgabeoptionen	
/o	Obj (BCOM.LIB)
Optimierungsoptionen	
/q	Geschwindigkeit

Ausgabeoptionen: Beim getrennten Kompilierverfahren benutzt QuickBASIC als Standardvorgabe die Option »Obj (BRUN.LIB)«, die wir für User-Libraries benötigen. Vergessen Sie bitte vorläufig den Parameter »/o«, mit dem die Option »Obj (BCOM.LIB)« gewählt wird. Wir besprechen diese Option später.

Optimierungsoptionen: Bei der Arbeit »im Dialog« mit QuickBASIC ist die Option »Geschwindigkeit« die Standard-Optimierungsvorgabe. Im getrennten Kompilierverfahren ist seltsamerweise statt »Geschwindigkeit« die Option »Umfang« die Standardvorgabe. Soll dennoch auf Geschwindigkeit optimiert werden, wählen Sie diese Option mit dem Schalter »/q«.

Beispiele:

1. *QB USER1 /E /Q;* : erzeugt die Objektdatei USER1.OBJ, in der ON ERROR-Anweisungen vorhanden sind und die auf Geschwindigkeit optimiert wird.
2. *QB USER1 /D /W /O;* : USER1.OBJ wird mit der Ausgabeoption »Obj (BCOM.LIB)« kompiliert (für die Einbindung in User-Libraries ungeeignet!); die Ereigniserfassung (Abfrage des Timers, des Joysticks etc.) ist eingeschaltet; »Austesten« ist eingeschaltet, das heißt, die Unterbrechung mit CTRL+BREAK und die Benutzung des Fehlersuch-Modus ist möglich.
3. *QB USER1 /D /Q;* : »Austesten« ist eingeschaltet; USER1.OBJ ist auf Geschwindigkeit optimiert.

Das dritte Beispiel benutzt die für unsere Zwecke nützlichsten Parameter. Das Programm USER1.BAS wird mit den üblichen Standardoptionen »Austesten« und »Geschwindigkeit« kompiliert.

Nun ein praktisches Beispiel: Sie erinnern sich an unsere beiden Demoprogramme USER1.BAS und USER2.BAS? Wir kompilieren diese Programme nun unter Verwendung des getrennten Kompilierverfahrens mit den Standardoptionen »Austesten« und »Geschwindigkeit«.

USER1.BAS und USER2.BAS kompilieren:

```
QB USER1 /D /Q;
QB USER2 /D /Q;
```

Nach diesen beiden Aufrufen befinden sich auf der Diskette die beiden Objektdateien USER1.OBJ und USER2.OBJ, die Sie nun wie gewohnt mit BUILDLIB zu einer User-Library zusammenbinden können.

Sie haben die Wahl: Objektdateien können Sie »im Dialog« mit QuickBASIC oder auf der Betriebssystem-Ebene erstellen. Richtig komfortabel wird das getrennte Kompilierverfahren aber erst, wenn eine vernünftige Dateiorganisation vorliegt und Batch-Dateien (Stapel-Dateien) verwendet werden.

4.5.1 Systemeinrichtung

Zur Erstellung einer User-Library benötigen wir außer QuickBASIC (QB.EXE) die beiden Dateien BUILDLIB.EXE und BRUN20.LIB.

Übrigens: BRUN20.LIB heißt in der Version 3.0 von QuickBASIC nun BRUN30.LIB!

Die einfachste Methode, den Zugriff auf diese Dateien zu ermöglichen, kennen Sie: Alle drei Dateien und die BASIC-Programme werden gemeinsam in einem Directory gespeichert, auf einer Diskette oder – wenn Sie eine Festplatte besitzen – in einem Subdirectory.

Dieser »Mischmasch« verschiedener Dateien wird jedoch schnell unübersichtlich. Eine getrennte Anordnung der QuickBASIC-Programmdateien und der BASIC-Programme ist weitaus übersichtlicher. Außerdem bekommen Sie Platzprobleme, wenn Sie mangels Festplatte mit einer Diskette arbeiten.

Eine getrennte Anordnung setzt voraus, daß Sie über eine Festplatte, zumindest aber über zwei Diskettenlaufwerke verfügen (ein nicht vorhandenes zweites Diskettenlaufwerk können Sie notfalls auch mit einer RAM-Disk simulieren). Auf der »Arbeitsdiskette« speichern wir außer QB.EXE unsere BASIC-Programme, auf der »Programmdiskette« alle benötigten Programmdateien von QuickBASIC.

Das folgende Schema zeigt die – meiner Meinung nach – optimale Aufteilung. Das Schema bezieht sich auf die Verwendung von zwei Diskettenlaufwerken A und B.

»Arbeitsdisk« in Laufwerk A	»Programmdisk« in Laufwerk B
– QB.EXE – BASIC-Programme (ROUTINEN.BAS etc.)	– QuickBASIC-Programmdateien: BUILDLIB.EXE, BRUN20.LIB, BRUN20.EXE, LINK.EXE – Batch-Dateien: DC.BAT und DL.BAT

In allen folgenden Kapiteln werde ich diese Dateiorganisation voraussetzen! Sollten Sie mit der Erzeugung der Objektdateien Schwierigkeiten haben: Alle größeren Module befinden sich nicht nur als BASIC-Programm im Verzeichnis MODULE, sondern zusätzlich bereits kompiliert im Verzeichnis OBJFILES.

Außer den wichtigsten Programmdateien von QuickBASIC (QB.EXE befindet sich bereits auf der Arbeitsdiskette) enthält die Programmdiskette in Laufwerk B die beiden Batch-Dateien DC.BAT und DL.BAT.

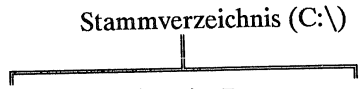
Richten Sie sich bitte diese beiden Disketten ein. Die Batch-Dateien finden Sie im Verzeichnis BATCHES.

Kopieren Sie zunächst:

1. auf die Arbeitsdiskette: QB.EXE und ROUTINEN.BAS
2. auf die Programmdiskette: BUILDLIB.EXE, BRUN20.LIB, BRUN20.EXE und LINK.EXE. Kopieren Sie auf die Programmdiskette zusätzlich die Dateien DC.BAT und DL.BAT.

Modifikationen für Festplattenbenutzer: Wenn Sie über eine Festplatte verfügen, gehen Sie bitte ins Stammverzeichnis (C:\) und richten Sie ein Unterverzeichnis QB ein. Dieses Verzeichnis verwenden Sie anstelle der Arbeitsdiskette. Es ist Ihr »Arbeitsverzeichnis«.

Richten Sie im Stammverzeichnis bitte ein weiteres Unterverzeichnis ein, SYSTEM20. Dieses Verzeichnis ist Ihr »Programmverzeichnis«. Ihre »Verzeichnisstruktur« sollte nun so aussehen:



Arbeitsverzeichnis (C:\QB) Programmverzeichnis (C:\SYSTEM20)

Kopieren Sie in das Verzeichnis SYSTEM20 statt DC.BAT und DL.BAT die Batch-Dateien PC.BAT und PL.BAT.

Wichtig: Ändern Sie die Datei AUTOEXEC.BAT, die beim Einschalten Ihres Rechners automatisch ausgeführt wird! Richten Sie einen Pfad auf die Programmdiskette ein (Laufwerk B beziehungsweise Verzeichnis

SYSTEM20). Fügen Sie in die bestehende AUTOEXEC.BAT-Datei die Anweisung ein:

Diskettenlaufwerke:	PATH B:
Festplatte:	PATH C:\SYSTEM20

Booten Sie nun bitte Ihren Rechner (CTRL+ALT+DEL), damit die geänderte AUTOEXEC.BAT-Datei ausgeführt und der Pfad eingerichtet wird.

Im folgenden gehe ich davon aus, daß A das Stammlaufwerk ist (Festplatte: QB ist das aktuelle Verzeichnis) und Sie einen Pfad zu Laufwerk B (Festplatte: Verzeichnis SYSTEM20) eingerichtet haben!

Kompilieren mit DC.BAT (PC.BAT)

Die Batch-Datei DC.BAT enthält alle nötigen Befehle, um bis zu neun BASIC-Programme nacheinander zu kompilieren.

```
If Exist %1.bas qb %1 /d /q;
If Exist %2.bas qb %2 /d /q;
If Exist %3.bas qb %3 /d /q;
If Exist %4.bas qb %4 /d /q;
If Exist %5.bas qb %5 /d /q;
If Exist %6.bas qb %6 /d /q;
If Exist %7.bas qb %7 /d /q;
If Exist %8.bas qb %8 /d /q;
If Exist %9.bas qb %9 /d /q;
copy *.obj b:
erase *.obj
```

In eine Batch-Datei können Sie beliebige DOS-Befehle schreiben. Beim Aufruf der Batch-Datei wird dieser »Befehls-Stapel« der Reihe nach abgearbeitet. Zahlen mit vorangestelltem Prozentzeichen sind die »Variablen« von Batch-Dateien. In »%1« wird der erste beim Aufruf angegebene Parameter gespeichert, in »%2« der zweite Parameter und so weiter. Mit *IF EXIST %NR* kann geprüft werden, ob der Parameter mit der betreffenden Nummer beim Aufruf angegeben wurde.

Beispiel: `DC ROUTINEN`

Beim Aufruf der Batch-Datei DC.BAT wurde ein Parameter angegeben: »ROUTINEN«. Dieser Parameter ist nun in »%1« enthalten. Weitere Parameter (%2, %3 etc.) existieren nicht.

»%1« enthält den Parameter »ROUTINEN«. Also kann die Anweisung

```
IF EXIST %1.BAS QB %1 /D /Q ;
```

mit

```
IF EXIST ROUTINEN.BAS QB ROUTINEN /D /Q ;
```

»übersetzt« werden.

Das Betriebssystem prüft, ob eine Datei ROUTINEN.BAS im aktuellen Verzeichnis existiert. Wenn ja, wird QuickBASIC mit den Parametern »/D« (Austesten) und »/Q« (Geschwindigkeit) aufgerufen. ROUTINEN.BAS wird mit diesen Optionen kompiliert und das Objektprogramm ROUTINEN.OBJ erzeugt. Falls ein zweiter Dateiname angegeben wurde, wird mit

```
IF EXIST %1.BAS QB %1 /D /Q ;
```

geprüft, ob auch diese Datei existiert. Wenn ja, wird das betreffende Programm ebenfalls kompiliert.

Diese Batch-Datei automatisiert den Kompiliervorgang. Sie können mit einem Schlag ein, zwei oder noch mehr Programme kompilieren lassen. Ein Beispiel für die automatisierte Kompilierung von drei Programmen:

```
DC ROUTINEN TEST DEMO
```

Probieren Sie es aus. Speichern Sie außer Routinen zwei weitere BASIC-Programme unter den Namen TEST und DEMO auf der Arbeitsdiskette. DC.BAT kompiliert die drei Programme in der angegebenen Reihenfolge.

Voraussetzung: Als aktuelles Verzeichnis muß das Arbeitsverzeichnis (Diskette: Laufwerk A / Festplatte: Verzeichnis QB) eingestellt sein. Sonst wird das Betriebssystem die dort gespeicherten BASIC-Programme nicht finden!

Die kompilierten Programme besitzen die Namenserweiterung .OBJ. Die in der Batch-Datei enthaltene Anweisung

```
COPY *.OBJ B:
```

kopiert alle erzeugten Objektdateien auf die Programmdiskette im Laufwerk B.

```
ERASE *.OBJ
```

löscht anschließend die Objektdateien im aktuellen Verzeichnis, also auf der Arbeitsdiskette. Sie werden auf der Arbeitsdiskette nicht mehr benötigt. Auf der Programmdiskette bleiben die erzeugten Objektdateien erhalten. Möglicherweise werden sie später noch einmal zur Erzeugung einer User-Library gebraucht.

Modifikationen für Festplatte: Verwenden Sie statt DC.BAT die Batch-Datei PC.BAT.

```
If Exist %1.bas qb %1 /d /q;
If Exist %2.bas qb %2 /d /q;
If Exist %3.bas qb %3 /d /q;
If Exist %4.bas qb %4 /d /q;
```

```
If Exist %5.bas qb %5 /d /q;
If Exist %6.bas qb %6 /d /q;
If Exist %7.bas qb %7 /d /q;
If Exist %8.bas qb %8 /d /q;
If Exist %9.bas qb %9 /d /q;
copy *.obj c:\system20
erase *.obj
```

Der einzige Unterschied zu DC.BAT besteht in der Angabe des Verzeichnisses SYSTEM20 anstelle des Diskettenlaufwerks B im COPY-Befehl. Bis auf diesen Unterschied sind beide Batch-Dateien miteinander identisch.

Denken Sie daran: Die Benutzung der Batch-Dateien setzt voraus, daß A das Stammlaufwerk ist (Festplatte: aktuelles Verzeichnis ist QB) und ein Pfad auf B eingerichtet wurde (Festplatte: Pfad auf SYSTEM20)! Das Gleiche gilt für die folgenden Batch-Dateien.

User-Library aufbauen mit DL.BAT (PL.BAT)

Die Batch-Datei DL.BAT bindet mehrere Objektdateien zu einer User-Library zusammen.

```
b:
buildlib %1 %2 %3 %4 %5 %6 %7 %8 %9,a:userlib;
a:
```

Die Objektdateien befinden sich nach dem Kompilieren mit DC.BAT auf der Diskette in Laufwerk B. Die Anweisung

```
B:
```

wechselt das Laufwerk. B ist nun das Stammlaufwerk. BUILDLIB wird aufgerufen, wobei die »Variablen« der Batch-Datei alle beim Aufruf angegebenen Dateinamen enthalten. BUILDLIB bindet die zugehörigen Objektdateien zu einer User-Library zusammen, die auf der Arbeitsdiskette (Laufwerk A) gespeichert wird. Zuletzt wird erneut das Laufwerk gewechselt. Das Stammlaufwerk ist wieder A.

Beispiel: `DL ROUTINEN TEST DEMO`

Dieser Aufruf der Batch-Datei bindet die drei zuvor mit DC.BAT erzeugten Objektdateien ROUTINEN.OBJ, TEST.OBJ und DEMO.OBJ zu einer User-Library zusammen. Die User-Library befindet sich danach auf der Arbeitsdiskette und kann beim Aufruf von QuickBASIC wie gewohnt mit *QB/L* resident geladen werden.

Modifikationen für Festplatte: Statt DL.BAT benutzen Sie PL.BAT, wenn Sie eine Festplatte verwenden.

```
cd c:\system20
buildlib %1 %2 %3 %4 %5 %6 %7 %8 %9,\qb\userlib;
cd c:\qb
```

In PL.BAT sind die Laufwerksnamen A und B durch die Namen der beiden Verzeichnisse QB und SYSTEM20 ersetzt.

Zusammenfassung:

1. Richten Sie sich bitte eine Arbeitsdiskette und eine Programmdiskette ein (Festplatte: ein Verzeichnis QB (Pfad: C:\QB) und ein Verzeichnis SYSTEM20 (Pfad: C:\SYSTEM20)).

2. Speichern Sie auf der Arbeitsdiskette die Datei QB.EXE und Ihre BASIC-Programme (in den folgenden Kapiteln wird vorläufig nur ROUTINEN.BAS benötigt).

3. Speichern Sie auf der Programmdiskette die Dateien BUILDLIB.EXE, BRUN20.LIB, LINK.EXE, BRUN20.EXE und BCOM20.EXE. Speichern Sie außerdem die Batch-Dateien DC.BAT und DL.BAT auf der Programmdiskette, die sich auf der Diskette zum Buch befinden (Festplatte: Speichern Sie statt dessen in SYSTEM20 die Dateien PC.BAT und PL.BAT). Alle Batch-Dateien finden Sie im Verzeichnis BATCHES.

4. Ändern Sie die Datei AUTOEXEC.BAT Ihrer Betriebssystem-Diskette. Geben Sie mit dem PATH-Befehl einen Pfad zur Programmdiskette (Festplatte: zum Programmverzeichnis) an. Ist in der AUTOEXEC.BAT-Datei bereits eine PATH-Anweisung enthalten, genügt es, diese entsprechend zu erweitern.

5. Um die beschriebenen Batch-Dateien zu benutzen, muß als Stammlaufwerk A eingestellt sein (Festplatte: aktuelles Verzeichnis QB).

6. Kompilieren: Rufen Sie DC.BAT (Festplatte: PC.BAT) auf und geben Sie beim Aufruf durch Leerzeichen getrennt die Namen der zu kompilierenden BASIC-Programme an (ohne die Erweiterung .BAS).

   Beispiel: `DC DEMO1 DEMO2 DEMO3`

   =>Kompilieren der Programme DEMO1, DEMO2 und DEMO3.

7. User-Library einrichten: Rufen Sie DL.BAT (Festplatte: PL.BAT) auf und geben Sie die Namen aller Objektdateien an, die in die User-Library eingebunden werden sollen (ohne Namenserweiterung .OBJ).

   Beispiel: `DL DEMO1 DEMO2 DEMO3`

   =>User-Library erzeugen, die DEMO1.OBJ, DEMO2.OBJ und DEMO3.OBJ enthält.

5

Eingaberoutine

Sie kennen alle erweiterten »Features« von QuickBASIC, Prozeduren, die Übergabe von Variablen, User-Libraries und so weiter. Dieses mehr theoretische Wissen vertiefen wir nun anhand mehrerer größerer Programmprojekte. Das erste ist eine Eingaberoutine. Die Routine ist auf der beiliegenden Diskette unter dem Namen INPUT.BAS gespeichert.

Eingaberoutinen werden für sehr unterschiedliche Zwecke benötigt. Für Einzeleingaben (»Bitte Vorname eingeben:«), für Eingaben in eine Maske oder sogar für komplette Bildschirm-Editoren.

Das heißt, eine gute Eingaberoutine muß sehr flexibel sein. INPUT.BAS erfüllt folgende Anforderungen:

1. Jede Eingabe findet in einer definierten Zone statt, die der Benutzer nicht willkürlich mit den Cursortasten verlassen kann. Die Zone wird durch die Bildschirm-Koordinaten der Startposition (Spalte und Zeile) und die Länge in Zeichen definiert.

2. Das aufrufende Programm kann in der Eingabezone einen Inhalt vorgeben. Beispiel: Der Benutzer will in einer Dateiverwaltung den Namen »Maier« in »Mayer« ändern. Das aufrufende Programm gibt den zu verbessernden Namen »Maier« vor; er erscheint in der Eingabezone. Der Benutzer muß nur noch »i« durch »y« ersetzen.

3. Es muß möglich sein, eine Eingabe außer mit RETURN auch mit anderen Tasten zu beenden. Beispiel: In einer Maske beendet CURSOR UNTEN die aktuelle Eingabe und bewegt den Cursor zum nächsten Feld; CURSOR OBEN beendet ebenfalls eine Eingabe und bewegt den Cursor zum vorhergehenden Feld.

   Fazit: Das aufrufende Programm teilt der Eingaberoutine mit, mit welchen Tasten eine Eingabe beendet wird. Die Eingaberoutine muß wiederum dem aufrufenden Programm mitteilen, mit welcher Taste die Eingabe beendet wurde. Nur so kann das aufrufende Programm entsprechend reagieren und den Cursor zum Beispiel zum vorhergehenden Feld bewegen, wenn die Eingabe mit CURSOR OBEN beendet wurde.

4. Es darf keinesfalls möglich sein, alle Zeichen einzugeben. Beispiel: Der Benutzer wird gefragt »Weitersuchen(j/n)?«. Im Beispiel darf die Eingaberoutine nur die Zeichen »j« und »n« akzeptieren. Alle anderen Tasten sind zu ignorieren.

   Das heißt, bei jedem Aufruf der Eingaberoutine wird eine Art »Zeichensatz« übergeben, der alle Zeichen enthält, die der Benutzer eingeben darf.

5. Hervorheben wichtiger Eingaben (Frage »Löschen (j/n)«) durch inverse Darstellung der Eingabezone.

6. Komfortable Editiermöglichkeiten (wortweises Bewegen des Cursors; Sprung zum Anfang/Ende der Eingabezone; Löschen bis zum Wortende; Eingabe komplett löschen).

7. Nützlich ist ein Flag, das dem aufrufenden Programm mitteilt, ob der Benutzer die Vorgabe veränderte. Zum Beispiel kann in einer Adressenverwaltung das Modul »Adressen ändern« an diesem Flag erkennen, ob eine in einer Eingabemaske vorgegebene Adresse wirklich verändert wurde. Wenn ja, wird sie auf der Diskette eingetragen. Fand keine Änderung statt, ist auch kein »Zurückschreiben« nötig.

Ein Nachteil der Flexibilität ist, daß beim Aufruf eine sehr umfangreiche Parameterliste anzugeben ist. Der Aufruf der Eingaberoutine:

```
Call lInput(E$,(Col),(Row),(Max),(Char$),(Mode),_
            (Back$),Insert,Last$,PosAkt,EditFlag)
```

Die Parameter in Klammern werden nach Wert übergeben. Die Eingaberoutine muß auf diese Werte nur lesend zugreifen. Alle anderen Parameter müssen nach Referenz übergeben werden! Diese Variablen liest die Eingaberoutine nicht nur, sie verändert sie auch, um Informationen an das aufrufende Programm zurückzugeben.

Parameter hin:

- <E$>: Der Inhalt von <E$> wird in der Eingabezone vorgegeben (leere Eingabezone gewünscht: E$="").
- <Col>/<Row>: Spalte und Zeile, an der die Eingabezone beginnt.
- <Max>: Länge der Eingabezone (=maximale Eingabelänge).
- <Char$>: alle Zeichen, die der Benutzer eingeben darf (gültige Eingabezeichen).
- <Mode>: Flag für die Bildschirmdarstellung (Mode=True => normale Darstellung; Mode=False => inverse Darstellung).

- <Back$>: Zeichen, die die Eingabe beenden.
- <Insert>: Flag für Einfüge-/Überschreibmodus (Insert=False => Überschreibmodus eingeschaltet; Insert=True => Einfügemodus eingeschaltet).
- <PosAkt>: Startposition des Cursors beim Aufruf der Eingaberoutine (normalerweise 1 = Anfang der Eingabezone).

Parameter zurück:

- <E$>: Nach Rückkehr aus der Eingaberoutine enthält <E$> den Feldinhalt. <E$> besitzt somit zwei Funktionen: In <E$> übergibt das aufrufende Programm einen vorgegebenen Feldinhalt und in <E$> übergibt die Eingaberoutine nach Beenden der Eingabe den aktuellen – eventuell von der Vorgabe abweichenden – Feldinhalt, also die Eingabe des Benutzers.
- <Insert>: Gibt nach der Rückkehr an, ob zuletzt der Einfüge- oder der Überschreibmodus eingeschaltet war (True => Einfügemodus; False => Überschreibmodus).
- <Last$>: String, der nach Rückkehr zum aufrufenden Programm das Zeichen enthält, mit dem die Eingabe beendet wurde.
- <PosAkt>: Letzte Cursorposition innerhalb der Eingabezone vor Beenden der Eingabe.
- <EditFlag>: Flag, das aussagt, ob der Inhalt der Eingabezone – also die Vorgabe – vom Benutzer verändert wurde (False => keine Änderungen vorgenommen; True => Inhalt der Eingabezone wurde verändert).

Vergessen Sie bitte vorläufig, daß die aktuellen Werte von <Insert> und <PosAkt> nach Beenden der Eingabe an das aufrufende Programm zurückgegeben werden. Bei Einzeleingaben sind diese Informationen für das aufrufende Programm bedeutungslos. Wir werden jedoch beide Parameter später noch bei komplexen Anwendungen wie der Steuerung einer Eingabemaske benötigen.

Beispiel für den Aufruf:

```
E$     = "Testeingabe"        'Vorgabe
Col    = 4: Row = 2           'Start der Zone (Spalte 4/Zeile 2)
Max    = 2Ø                   'Länge der Eingabezone (2Ø Zeichen)
Char$  = Alpha$               'nur numerische Eingaben zulassen
Back$  = zReturn$ + zEsc$     'Beenden mit RETURN und ESC möglich
Mode   = True                 'normale Darstellung der Eingabezone
Insert = True                 'Einfügemodus eingeschaltet
PosAkt = 1                    'Startposition: Cursor auf 1.Zeichen
```

```
Call lInput(E$,(Col),(Row),(Max),(Char$),(Mode),_
            (Back$),Insert,Last$,PosAkt,EditFlag)
```

Dieser Aufruf setzt voraus, daß einige Variablen als global deklariert wurden. <Alpha$> enthält alle alphanumerischen Zeichen, <zReturn$> und <zEsc$> sind die Zeichen mit den Codes 13 und 27. Alle drei Variablen sind in unseren grundlegenden Dateien COMDEF.BAS und INIT.BAS enthalten.

Sollte Sie diese lange Vorbereitung erschrecken: Natürlich können Sie alle Parameter, die nicht per Referenz übergeben werden, auch direkt übergeben:

```
E$ = "Testeingabe"
Call lInput(E$, 4, 2, 20, Alpha$, True, zReturn$ + zEsc$, True, Last$, 1,_
   EditFlag)
```

Die ausführliche Form sollte Ihnen nur klarmachen, welche Bedeutung die einzelnen Parameter besitzen.

Nach dem Aufruf erscheint in Eingabezone, die ab Spalte 4 von Zeile 2 beginnt und 20 Zeichen lang ist, die Vorgabe »Testeingabe« und kann nun editiert werden.

```
Testeingabe
^  (Eingabezone)   ^
...
...
...
```

Die Editiermöglichkeiten:

- CURSOR RECHTS/LINKS: Cursor um eine Spalte nach rechts/links bewegen
- HOME/END: Cursor auf den Anfang/das Ende der Eingabezone setzen
- CTRL+RECHTS/CTRL+LINKS: zum nächsten / vorhergehenden Wort springen
- DEL: das Zeichen an der aktuellen Cursorposition löschen
- BACKSPACE: das Zeichen links vom Cursor löschen

- CTRL+t: alle Zeichen ab der aktuellen Cursorposition bis zum Anfang des nächsten Wortes löschen
- CTRL+y: Eingabezone komplett löschen
- INS: zwischen dem Einfüge- und dem Überschreibmodus umschalten

Sie sehen, die Eingaberoutine wird sehr komfortable Editiermöglichkeiten bieten. Die Kombinationen CTRL+t (Wort löschen) und CTRL+y (Zeile löschen) sind an WordStar und kompatible Editoren angelehnt und den meisten unter Ihnen sicher vertraut.

5.1 Programmentwicklung

LINPUT besteht aus drei Teilen:

1. Einem Initialisierungsteil, der verschiedenen Variablen Ausgangswerte zuweist.
2. Der Eingabeschleife, die auf eine Taste wartet und je nach Taste zu einer entsprechenden Prozedur verzweigt.
3. Einer Anzahl von Prozeduren, die je nach gedrückter Taste den Eingabestring <E$> oder die Cursorposition <PosAkt> verändern.

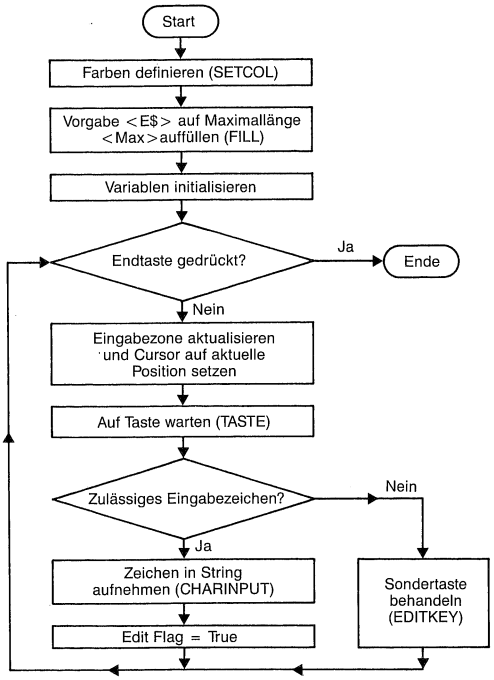


Bild 1 *Initialisierung und Eingabeschleife*

```
'       Line-Input-Routine
'       ------------------
'       Funktion: Eingabe in definierter Zone
'       Aufruf  : Call LInput(E$, (Col), (Row), (Max),(Char$), (Mode),_
'                             (Back$), Insert, PosAkt, Last$, EditFlag)
'       Hin     : E$       : Eingabevorgabe
'                 Col/Row  : Start Eingabezone
'                 Max      : Maximale Zeichenanzahl
'                 Char$    : Zulässige Zeichen
'                 Mode     : Ø=StandardColors/1=invers
'                 Back$    : Endezeichen
'                 Insert   : Ø=Einfügen/1=Überschreiben
'                 PosAkt   : Startposition des Cursors in der Zone
'       Zurück  : E$       : Eingabe
'                 Insert   : False=Überschreiben/True=Einfügen
'                 PosAkt   : Letzte Cursorposition in der Zone
'                 Last$    : Endezeichen
'                 EditFlag : False=keine Änderungen/True=Änderungen


Sub LInput(E$, Col, Row, Max, Char$, Mode,_
           Back$, Insert, PosAkt, Key$, EditFlag) Static
```

```
'       * Initialisierung *
  Call SetCol((Mode))
  Call Fill(E$,(Max))
  EditFlag = False
  Key$ = " "

'       * Eingabe *
  While Instr(Back$, Key$) = Ø
    Locate Row, Col, Ø: Print E$; :Locate Row, Col + PosAkt - 1, 1
    Call Taste(Key$, Escape)
    If Instr(Char$, Key$) <> Ø And Escape = False Then
      Call CharInput(E$, (Key$), (Max), PosAkt, (Insert))
      EditFlag = True
    Else
      Call EditKey((Key$), E$, (Max), PosAkt, EditFlag, Insert)
    End If
  Wend
  Call Compress(E$)
  Call SetCol(True)
  Locate ,,Ø
End Sub
```

Zuerst wird mit der Prozedur SETCOL die Hintergrund- und die Zeichenfarbe bestimmt, abhängig vom übergebenen Wert <Mode> (Mode=0 => normale / Mode=1 => inverse Darstellung). Mit der FILL-Prozedur (in der Routinen-Sammlung enthalten) wird die Vorgabe <E$> mit Leerzeichen auf die maximale Eingabelänge aufgefüllt.

<Key$> enthält die zuletzt gedrückte Taste und wird als »Leerstring« initialisiert. <EditFlag> erhält den Startwert <False> ((noch) keine Änderungen vorgenommen).

Nun beginnt die eigentliche Eingabeschleife. Das Programm wartet auf eine Taste und verzweigt zu einer Prozedur, die das betreffende Kommando ausführt (CURSOR RECHTS, DEL etc.). Die in der Schleife enthaltenen Anweisungen werden wiederholt, solange der Benutzer keines der in <Back$> enthaltenen Endezeichen eingibt (*While Instr(Back$, Key$) = 0*).

Nach jedem Tastendruck muß die Bildschirmausgabe aktualisiert werden, also der Inhalt der Eingabezone und die Cursorposition. Der Cursor wird auf den durch <Col> und <Row> definierten Beginn der Zone gesetzt und der Eingabestring <E$> komplett ausgegeben. Danach wird der Cursor auf die aktuelle Spalte gesetzt, die sich durch Addition der Startspalte <Col> und der aktuellen Positionsnummer <PosAkt> ergibt.

Die Prozedur TASTE (in der Routinen-Sammlung) wird aufgerufen. Die gedrückte Taste ist anschließend in <Key$> enthalten. Ist die Taste keine Sondertaste (Escape = False) und eines der zulässigen Eingabezeichen (*If Instr(Char$, Key$) <> 0 And Escape = False*), wird die Prozedur CHARINPUT aufgerufen. CHARINPUT fügt das Zeichen in <E$> ein und

aktualisiert die Cursorposition. <EditFlag> wird auf <True> gesetzt, da eine Veränderung der Vorgabe stattfand.

Wenn CHARINPUT nicht aufgerufen wird, wird statt dessen EDITKEY aufgerufen. Diese Prozedur vergleicht die Taste mit den zulässigen Editiertasten. Bei einer Übereinstimmung wird eine Prozedur aufgerufen, die das betreffende Kommando ausführt, zum Beispiel den Cursor nach rechts oder links bewegt.

Diese Eingabeschleife wird erst verlassen, wenn der Benutzer eine in <Back$> enthaltene Endetaste drückte. COMPRESS entfernt überflüssige Leerzeichen hinter der »echten« Eingabe. Mit SETCOL wird die Normaldarstellung eingeschaltet und mit *LOCATE „0* der Cursor wieder ausgeschaltet.

LINPUT gibt fünf Parameter an das aufrufende Programm zurück:

1. <E$>: die Eingabe
2. <Key$>: zuletzt gedrückte Taste (=Endetaste)
3. <EditFlag>: False => Vorgabe nicht verändert; True => die Vorgabe wurde verändert
4. <Insert>: False => Überschreibmodus eingeschaltet; True => Einfügemodus eingeschaltet
5. <PosAkt>: die letzte Position des Cursors innerhalb der Eingabezone

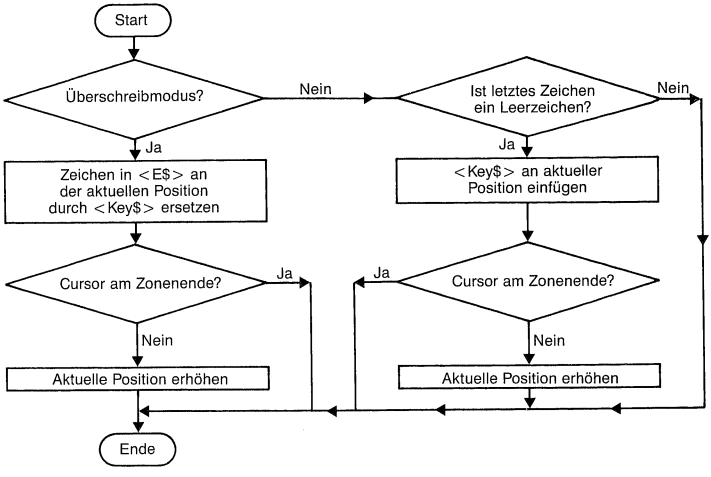


Bild 2 *Eingabe eines zulässigen Zeichens*

```
'       CharInput
'       ---------
'       Funktion: Eingabe eines zulässigen Zeichens
'       Aufruf  : Call CharInput(E$, (Key$), (Max), PosAkt, (Insert))
'       Hin     : E$     : Eingabe
'                 Key$   : Zeichen
'                 Max    : Maximale Zeichenanzahl
'                 PosAkt : Aktuelle Position in der Eingabezone
'                 Insert : False=Überschreiben/True=Einfügen
'       Zurück  : E$     : Eingabe aktualisiert
'                 PosAkt : Aktualisierte Position

Sub CharInput(E$, Key$, Max, PosAkt, Insert) Static
  If Insert = False Then
    Mid$(E$, PosAkt, 1) = Key$
    If PosAkt < Max Then PosAkt = PosAkt + 1
  Else
    If Right$(E$, 1) = " " Then
      E$=Left$(E$, PosAkt - 1) + Key$ + Mid$(E$, PosAkt, Len(E$) - PosAkt)
      If PosAkt < Max Then PosAkt = PosAkt + 1
    End If
  End If
End Sub
```

Zu Beginn von CHARINPUT wird geprüft, ob der Überschreibmodus eingeschaltet ist (*If Insert = False Then ...*). Wenn ja, wird in <E$> das Zeichen Nummer <PosAkt> – das Zeichen an der aktuellen Eingabeposition – mit dem soeben eingegebenen Zeichen <Key$> überschrieben.

Da sich der Cursor nach der Eingabe eines Zeichens um eine Spalte nach rechts bewegen soll, wird <PosAkt> um 1 erhöht. Mit einer Ausnahme: Wenn bereits das Ende der Eingabezone erreicht und somit die Bedingung *If PosAkt < Max* nicht erfüllt ist, bleibt <PosAkt> unverändert.

Bei eingeschaltetem Einfügemodus muß das Zeichen <Key$> an der aktuellen Eingabeposition <PosAkt> in <E$> eingefügt werden. Das heißt, alle nachfolgenden Zeichen im String werden nach rechts verschoben. Um ein »Herausfallen« des letzten Zeichens zu vermeiden, wird das Einfügen nur gestattet, wenn das letzte Zeichen ein Leerzeichen ist (*If Right$(E$, 1) = " "*).

Ist diese Bedingung erfüllt, wird <E$> neu zusammengesetzt. <E$> wird aus dem linken Teil bis zur aktuellen Eingabeposition, dem einzufügenden Zeichen <Key$> und den Zeichen rechts von der Eingabeposition gebildet. Danach wird ebenso wie im Überschreibmodus der Positionszähler um 1 erhöht, wenn nicht bereits das Ende der Eingabezone erreicht ist.

CHARINPUT verändert den Eingabestring <E$> und den Positionszähler <PosAkt>. Die Ausgabe auf dem Bildschirm und die Position des Cursors bleibt zunächst unverändert. Die nötige Aktualisierung auf dem Bildschirm übernimmt die Eingabeschleife nach Rückkehr aus CHARINPUT. Der

String <E$> wird komplett ausgegeben und der Cursor auf die neue aktuelle Position gesetzt.

Initialisierung der Farben

SETCOL ist eine »triviale« Prozedur. Der übergebene Parameter <Mode> ist entweder <True> (normale Darstellung) oder <False> (inverse Darstellung).

```
'       Color bestimmen
'       ---------------
'       Funktion: Bestimmt Farben für nächste Ausgaben
'       Aufruf  : Call SetCol((Mode))
'       Hin     : Mode : True=normal/False=invers

Sub SetCol(Mode) Static
  If Mode = True  Then Color CharCol, GroundCol
  If Mode = False Then Color GroundCol, CharCol
End Sub
```

Abhängig vom Zustand des Flags <Mode> wird mit der COLOR-Anweisung die Normaldarstellung oder die inverse Darstellung eingeschaltet.

- Normaldarstellung: Zeichen in Standard-Zeichenfarbe / Hintergrund in Standard-Hintergrundfarbe darstellen
- inverse Darstellung: Zeichen in Standard-Hintergrundfarbe / Hintergrund in Standard-Zeichenfarbe darstellen

Die Behandlung von Sondertasten

Die Prozedur EDITKEY behandelt Sondertasten. Ebenso wie CHARINPUT ändert auch diese Prozedur nur <E$> und den Positionszähler <PosAkt> und nicht die Bildschirmausgabe. Der Bildschirm wird nach Rückkehr aus dieser Prozedur von der Eingabeschleife aktualisiert.

Tasten, die keine besonderen programmtechnischen Probleme stellen, werden in EDITKEY sofort behandelt. Schwierigere Kommandos wie CTRL+T (=Löschen bis zum Wortende) übernehmen separate Prozeduren. Am Ende von EDITKEY wird <EditFlag> auf <True> gesetzt, wenn die betreffende Taste den Inhalt der Eingabezone veränderte.

```
'       Edit-Keys
'       ---------
'       Funktion: Cursorposition verändern
'       Aufruf  : Call EditKey((Key$), E$, (Max), PosAkt, EditFlag, Insert)
'       Hin     : Key$     : Taste
'                 E$       : Eingabe
'                 Max      : max.Zeichenanzahl
'                 PosAkt   : aktuelle Position
'                 Insert   : Ø=Einfügen/1=Überschreiben
```

```
'       Zurück  : E$       : aktualisierte Eingabe
'                 PosAkt   : aktualisierte Position
'                 EditFlag : False=keine Änd./True=Änderungen
'                 Insert   : Ø=Einfügen/1=Überschreiben

Sub EditKey(Key$, E$, Max, PosAkt, EditFlag, Insert) Static
  If Key$ = zIns$       Then Insert = Insert Xor 1
  If Key$ = zRight$     Then If PosAkt < Max Then PosAkt = PosAkt + 1
  If Key$ = zLeft$      Then If PosAkt > 1   Then PosAkt = PosAkt - 1
  If Key$ = zHome$      Then PosAkt = 1
  If Key$ = zEnd$       Then PosAkt = Max
  If Key$ = zDel$       Then E$ = Left$(E$, PosAkt - 1) +_
                                 Right$(E$, Max - PosAkt) + " "
  If Key$ = zBack$      Then Call CharBack(E$, (Max), PosAkt)
  If Key$ = zCtrlRight$ Then Call CharCtrlRight(E$, (Max), PosAkt)
  If Key$ = zCtrlLeft$  Then Call CharCtrlLeft(E$, (Max), PosAkt)
  If Key$ = zCtrlT$     Then Call CharCtrlT(E$, (Max), PosAkt)
  If Key$ = zCtrlY$     Then E$ = Space$(Max)
  If Key$ = zCtrlY$ Or Key$ = zDel$ Or_
    Key$ = zBack$ Or Key$ = zCtrlT$ Then EditFlag = True
End Sub
```

Die Behandlung der Taste INS ist ein wenig trickreich. INS schaltet zwischen dem Einfüge- und dem Überschreibmodus um. Den aktuellen Modus zeigt das Flag <Insert> an. Bei Betätigung der Taste INS muß der Zustand dieses Flags umgedreht werden. Die übliche Vorgehensweise:

```
If Insert = True Then Insert = False Else Insert = True
```

Die gleiche Wirkung wird einfacher mit der logischen Verknüpfung XOR erzielt (XOR = Exklusives ODER, umgangssprachlich »entweder oder«).

Die Variablen <True> und <False> besitzen die Werte 0 und 1. Bei der Anweisung *Insert = Insert Xor 1* können wir daher zwei Fälle unterscheiden, je nach aktuellem Zustand des Flags <Insert>:

1. Wenn <Insert> den Wert <False> besitzt, also den Wert 0, lautet die Verknüpfung 0 Xor 1 mit dem Ergebnis 1.

2. Wenn <Insert> den Wert <True> besitzt, also den Wert 1, lautet die Verknüpfung 1 Xor 1 mit dem Ergebnis 0.

In beiden Fällen wird der Flagzustand umgedreht, aus 0 (False) wird 1 (True) beziehungsweise aus 1 (True) wird 0 (False).

Geradezu trivial ist die Behandlung der Cursortasten. Der String <E$> wird durch die Betätigung einer Cursortaste nicht beeinflußt, sondern nur der Positionszähler <PosAkt> verändert.

1. CURSOR RECHTS (<zRight$>): Wenn das Ende der Eingabezone noch nicht erreicht ist (*If PosAkt < Max*), wird der Positionszähler <PosAkt> um eins erhöht.

2. CURSOR LINKS (<zLeft$>): Wenn sich der Cursor nicht am Anfang der Eingabezone befindet (*If PosAkt > 1*), wird der Positionszähler <PosAkt> um eins vermindert.

3. HOME (<zHome$>): <PosAkt> erhält den Wert 1 (Cursor auf erstem Zeichen der Eingabezone).

4. END (<zEnd$>): <PosAkt> wird mit der Wert <Max> gleichgesetzt, also die Länge der Eingabezone. Das heißt, nach der Aktualisierung der Cursorposition in der Eingabeschleife befindet sich der Cursor am Ende der Eingabezone.

Die Taste DEL erfordert ein wenig »String-Basteln«. DEL löscht das Zeichen an der aktuellen Cursorposition <PosAkt>. <E$> wird aus dem linken Teil bis vor dieser Position (*Left$(E$, PosAkt – 1)*) und dem rechten Teil nach dieser Position (*Right$(E$, Max – PosAkt)*) zusammengesetzt. Da nun ein Zeichen fehlt, wird an <E$> ein Leerzeichen angehängt, um die Stringlänge konstant zu halten.

Etwas komplexer ist die Prozedur CHARBACK, die die Taste BACKSPACE behandelt. BACKSPACE löscht das Zeichen links neben dem Cursor, rückt alle nachfolgenden Zeichen auf und bewegt den Cursor um eine Spalte nach links.

```
Sub CharBack(E$, Max, PosAkt) Static
  If PosAkt > 1 Then
    E$ = Left$(E$, PosAkt - 2) + Mid$(E$, PosAkt) + " "
    PosAkt = PosAkt - 1
  End If
End Sub
```

Das Zeichen links vom Cursor kann nur gelöscht werden, wenn sich der Cursor nicht bereits auf dem ersten Zeichen befindet und die Bedingung *If PosAkt > 1* erfüllt ist.

<E$> wird neu zusammengesetzt, aus dem linken Teil der Eingabe bis vor dem zu löschenden Zeichen (*Left$(E$, PosAkt – 2)*) und dem rechten Teil ab der aktuellen Cursorposition. Um die Stringlänge trotz gelöschtem Zeichen konstant zu halten, wird ein Leerzeichen an das Stringende angehängt. Zum Abschluß wird der Positionszähler um eins vermindert.

Wortweise springen/Löschen bis zum Wortende

Die Prozeduren CHARCTRLRIGHT und CHARCTRLLEFT sind die schwierigsten Teile der gesamten Eingaberoutine. Mit CTRL+RECHTS wird der Cursor zum Anfang des nächsten Wortes bewegt, mit CTRL+LINKS zum vorigen Wort. Beide Prozeduren sollen ebenso funktionieren wie der Editor von QuickBASIC. Das heißt, nicht nur Wortzwischen-

räume aus Leerzeichen, sondern auch andere Trennzeichen sollen übersprungen werden.

Die folgenden Beispiele demonstrieren die genaue Arbeitsweise. Das Zeichen » ^ « soll die Cursorposition vor beziehungsweise nach dem »Wortsprung« anzeigen.

Beispiele:

1. CTRL + RECHTS:

```
a)  Vorher:  Mit   freundlichen ... Grüßen
             ^
    Nachher: Mit   freundlichen ... Grüßen
                   ^
b)  Vorher:  Mit   freundlichen ... Grüßen
                       ^
    Nachher: Mit   freundlichen ... Grüßen
                                    ^
c)  Vorher:  Mit   freundlichen ... Grüßen
                                        ^
    Nachher: Mit   freundlichen ... Grüßen
                                        ^
```

2. CTRL + LINKS:

```
a)  Vorher:  Mit   freundlichen ... Grüßen
                       ^
    Nachher: Mit   freundlichen ... Grüßen
                   ^
b)  Vorher:  Mit   freundlichen ... Grüßen
                 ^
    Nachher: Mit   freundlichen ... Grüßen
             ^
c)  Vorher:  Mit   freundlichen ... Grüßen
                                    ^
    Nachher: Mit   freundlichen ... Grüßen
                   ^
```

Wie diese Beispiele zeigen, unterscheiden sich die Funktionen »Wort nach rechts« und »Wort nach links« ein wenig voneinander. »Wort nach rechts« springt immer zum nächsten Wort. »Wort nach links« springt dagegen nicht immer zum vorhergehenden Wort. Zum vorhergehenden Wort wird nur gesprungen, wenn sich der Cursor am Anfang eines Wortes (Beispiel 2c) oder in einem Wortzwischenraum (Beispiel 2b) befindet. Ist der Cursor dagegen mitten in einem Wort (Beispiel 2a), wird zum ersten Zeichen dieses Wortes gesprungen.

Ob Sie bei QuickBASIC, SideKick, Turbo Pascal oder WordStar »wortweise springen«: Alle »vernünftigen« Editoren arbeiten auf die geschilderte Weise.

Wie die Beispiele zeigen, können als Trennzeichen zwischen Wörtern außer Leerzeichen auch Punkte vorkommen. Ebenso sind Klammern, Kommata

und andere »Sonderzeichen« denkbar. Eine sinnvolle Zusammenstellung von Trennzeichen enthält der String <Terminator$>. <Terminator$> ist in COMDEF.BAS als globale Variable deklariert und wird in INIT.BAS mit folgender Zeichenkette initialisiert:

```
Terminator$ = ",;.:!§$%&/()=?<>#^+-*_ "
```

Sollten Ihnen zusätzliche Trennzeichen einfallen, erweitern Sie bitte <Terminator$> in der Datei INIT.BAS entsprechend.

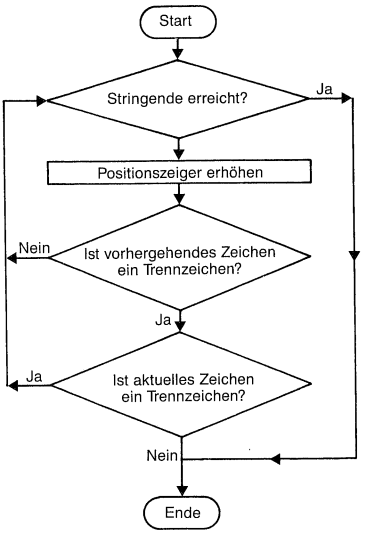


Bild 3 *Sprung nach rechts*

1. Zuerst wird geprüft, ob sich der Cursor am Ende der Eingabezone befindet, ob <PosAkt> auf das Stringende zeigt. Wenn ja, ist ein Sprung nach rechts nicht möglich. Die Prozedur wird verlassen.
2. Sonst wird der Positionszähler um 1 erhöht und geprüft, ob der Anfang eines Wortes erreicht ist. Dies ist der Fall, wenn das Zeichen vor der aktuellen Position *(PosAkt – 1)* ein Trennzeichen und das Zeichen an der aktuellen Position (PosAkt) kein Trennzeichen ist.
3. Ist noch kein Wortanfang erreicht, wiederholt sich der gesamte Vorgang ab der Prüfung auf das Erreichen des Stringendes.

```
Sub CharCtrlRight(E$, Max, PosAkt) Static
  While PosAkt < Max
    PosAkt = PosAkt + 1
    If Instr(Terminator$, Mid$(E$, PosAkt - 1, 1)) <> Ø_
       And Instr(Terminator$, Mid$(E$, PosAkt, 1)) = Ø Then Exit Sub
  Wend
End Sub
```

Der Programmablauf wird mit einer WHILE-Schleife umgesetzt. Die Anweisungen in der Schleife werden wiederholt, bis die Bedingung *PosAkt < Max* nicht mehr erfüllt, also das Stringende erreicht ist.

Der Positionszähler <PosAkt> wird um 1 erhöht. Dann folgt die Prüfung auf einen erreichten Wortanfang. Mit der INSTR-Funktion wird geprüft, ob das Zeichen unmittelbar vor der aktuellen Position (*PosAkt – 1*) ein Trennzeichen ist (Trennzeichen = Zeichen, das in <Terminator$> enthalten ist).

Nur dann, wenn diese Bedingung erfüllt ist und gleichzeitig das Zeichen an der aktuellen Position <PosAkt> kein (!) Trennzeichen ist, wird die Schleife mit EXIT SUB verlassen. Wie im PAP wird die Schleife fortgesetzt, wenn eine der beiden Bedingungen nicht erfüllt ist.

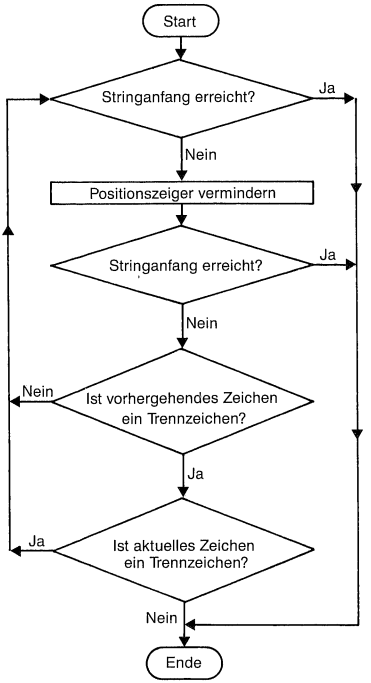


Bild 4 *Sprung nach links*

Der Positionszähler <PosAkt> wird diesmal natürlich vermindert und nicht wie beim Sprung nach rechts erhöht. Der Cursor soll sich in Richtung Stringanfang bewegen!

```
Sub CharCtrlLeft(E$, Max, PosAkt) Static
  While PosAkt > 1
    PosAkt = PosAkt - 1
    If PosAkt = 1 Then Exit Sub
    If Instr(Terminator$, Mid$(E$, PosAkt - 1, 1)) <> Ø_
       And Instr(Terminator$, Mid$(E$, PosAkt, 1)) = Ø Then Exit Sub
  Wend
End Sub
```

Die Schleife wird verlassen, wenn <PosAkt> nicht mehr größer ist als 1, das heißt, wenn das erste Zeichen erreicht wurde. In der Schleife wird <PosAkt> um eins vermindert und auf die gleiche Weise wie beim Sprung nach rechts geprüft, ob der Anfang eines Wortes erreicht ist.

Vielleicht erscheint es Ihnen ungewöhnlich, daß nach dem Vermindern des Positionszählers erneut geprüft wird, ob der Stringanfang erreicht ist

(*If PosAkt = 1 Then Exit Sub*). Diese Zusatzprüfung ist nur wegen einem Spezialfall nötig.

Angenommen, der Cursor befindet sich beim Kommando CTRL+LINKS auf dem zweiten Zeichen der Eingabe. <PosAkt> besitzt den Wert 2. Nun wird die Prozedur CHARCTRLLEFT aufgerufen. <PosAkt> ist größer als 1 und damit die Bedingung *While PosAkt > 1* erfüllt – die Schleifenanweisungen werden ausgeführt. <PosAkt> wird um 1 vermindert und hat den neuen Wert 1.

Und nun wird das Programm »abstürzen«! Mit *Instr(Terminator$, Mid$(E$, PosAkt – 1, 1))* wird das Zeichen Nummer *PosAkt – 1* mit einem Trennzeichen verglichen. *PosAkt – 1* ergibt jedoch 0! Die MID$-Funktion soll auf das Zeichen Nummer 0 in <E$> zugreifen. Dieses Zeichen existiert natürlich nicht und das Programm wird abstürzen (oder Sie erhalten eine Laufzeit-Fehlermeldung, wenn Sie mit »Austesten« kompiliert haben).

Nur wegen diesem einen Spezialfall (Sprung nach links vom zweiten Zeichen der Eingabe aus) ist die zusätzliche Prüfung auf den Stringanfang notwendig!

Bis zum nächsten Wort löschen

Mit CTRL+t werden alle Zeichen ab der aktuellen Cursorposition bis zum Anfang des nächsten Wortes gelöscht.

```
Vorher:  Mit   freundlichen ... Grüßen
                     ^
Nachher: Mit   freundGrüßen
                     ^
```

Das heißt, wir bauen <E$> aus dem linken Teil bis zum Zeichen vor der aktuellen Cursorposition und aus dem rechten Teil ab dem Beginn des nächsten Wortes neu zusammen. Um herauszubekommen, wo das nächste Wort beginnt, benutzen wir die Prozedur CHARCTRLRIGHT; wir simulieren einen Wortsprung nach rechts.

```
Sub CharCtrlT(E$, Max, PosAkt) Static
  PosNeu = PosAkt
  Call CharCtrlRight(E$, (Max), PosNeu)
  E$ = Left$(E$, PosAkt - 1) + Mid$(E$, PosNeu)
  Call Fill(E$, Max)
End Sub
```

Die aktuelle Position <PosAkt> wird in die Variable <PosNeu> kopiert. Mit <PosNeu> als Parameter, der die aktuelle Cursorposition angibt, wird CHARCTRLRIGHT aufgerufen. Die Prozedur sucht den Anfang des nächsten Wortes und übergibt ihn in <PosNeu>.

<E$> wird nun aus dem linken Teil bis zum Zeichen vor der echten aktuellen Cursorposition <PosAkt> und dem mittleren Teil ab der Position des nächsten Wortes <PosNeu> gebildet.

<E$> ist nun kürzer als zuvor. Wir können <E$> diesmal jedoch nicht einfach durch Anhängen eines Leerzeichens auf die Maximallänge auffüllen. Wir wissen nicht, wieviele Zeichen gelöscht wurden. Daher rufen wir FILL (in der Routinen-Sammlung) auf. Wir übergeben <E$> und die gewünschte Länge <Max> und erhalten den mit Leerzeichen aufgefüllten Eingabestring zurück.

Komplette Eingabe löschen

CTRL+y löscht die komplette Eingabezone. Ein Befehl genügt:

```
If Key$ = zCtrlY$      Then E$ = Space$(Max)
```

<E$> werden <Max> Leerzeichen zugewiesen. Die Länge von <E$> bleibt konstant und entspricht der Länge der Eingabezone.

5.2 Listing der Eingaberoutine (INPUT.BAS)

Die Eingaberoutine befindet sich unter dem Namen INPUT.BAS auf der Diskette zum Buch. Und zwar im Verzeichnis MODULE, wie auch alle weiteren Programme, die wir noch entwickeln. INPUT.BAS ist bereits zur Einbindung in die User-Library vorbereitet: Am Programmanfang befindet sich die Anweisung $INCLUDE: 'comdef.bas'. Diese Anweisung zur Einbindung der Deklarationsdatei darf vor der Kompilierung als eigenständiges Objektmodul keinesfalls vergessen werden. Die verschiedenen Prozeduren verwenden viele der in COMDEF.BAS als global deklarierten Variablen (unter anderem <Terminator$> und alle »Tastenvariablen«)! Ohne Einbindung von COMDEF.BAS sind diese Variablen in unserem Modul nicht bekannt.

```
'       **********************
'       *   Eingaberoutine   *
'       **********************

'       *** (C) Said Baloui, 1987 ***

REM $INCLUDE: 'comdef.bas'
'-----------------------------------------------------------------

'       Color bestimmen
'       ---------------
'       Funktion: Bestimmt Farben für nächste Ausgaben
'       Aufruf  : Call SetCol((Mode))
'       Hin     : Mode : True=normal/False=invers

Sub SetCol(Mode) Static
  If Mode = True  Then Color CharCol, GroundCol
  If Mode = False Then Color GroundCol, CharCol
End Sub
'-----------------------------------------------------------------

'       CharInput
'       ---------
'       Funktion: Eingabe eines zulässigen Zeichens
'       Aufruf  : Call CharInput(E$, (Key$), (Max), PosAkt, (Insert))
'       Hin     : E$     : Eingabe
'                 Key$   : Zeichen
'                 Max    : Maximale Zeichenanzahl
'                 PosAkt : Aktuelle Position in der Eingabezone
'                 Insert : False=Überschreiben/True=Einfügen
'       Zurück  : E$     : Eingabe aktualisiert
'                 PosAkt : Aktualisierte Position

Sub CharInput(E$, Key$, Max, PosAkt, Insert) Static
  If Insert = False Then
    Mid$(E$, PosAkt, 1) = Key$
    If PosAkt < Max Then PosAkt = PosAkt + 1
  Else
    If Right$(E$, 1) = " " Then
      E$=Left$(E$, PosAkt - 1) + Key$ + Mid$(E$, PosAkt, Len(E$) - PosAkt)
      If PosAkt < Max Then PosAkt = PosAkt + 1
    End If
  End If
End Sub
'-----------------------------------------------------------------

'       Edit-Keys
'       ---------
'       Funktion: Cursorposition verändern
'       Aufruf  : Call EditKey((Key$), E$, (Max), PosAkt, EditFlag, Insert)
'       Hin     : Key$     : Taste
'                 E$       : Eingabe
'                 Max      : max.Zeichenanzahl
'                 PosAkt   : aktuelle Position
'                 Insert   : Ø=Einfügen/1=Überschreiben
```

```
'       Zurück  : E$       : aktualisierte Eingabe
'                 PosAkt   : aktualisierte Position
'                 EditFlag : False=keine Änd./True=Änderungen
'                 Insert   : Ø=Einfügen/1=Überschreiben

Sub EditKey(Key$, E$, Max, PosAkt, EditFlag, Insert) Static
  If Key$ = zIns$       Then Insert = Insert Xor 1
  If Key$ = zRight$     Then If PosAkt < Max Then PosAkt = PosAkt + 1
  If Key$ = zLeft$      Then If PosAkt > 1   Then PosAkt = PosAkt - 1
  If Key$ = zHome$      Then PosAkt = 1
  If Key$ = zEnd$       Then PosAkt = Max
  If Key$ = zDel$       Then E$ = Left$(E$, PosAkt - 1) +_
                                 Right$(E$, Max - PosAkt) + " "
  If Key$ = zBack$      Then Call CharBack(E$, (Max), PosAkt)
  If Key$ = zCtrlRight$ Then Call CharCtrlRight(E$, (Max), PosAkt)
  If Key$ = zCtrlLeft$  Then Call CharCtrlLeft(E$, (Max), PosAkt)
  If Key$ = zCtrlT$     Then Call CharCtrlT(E$, (Max), PosAkt)
  If Key$ = zCtrlY$     Then E$ = Space$(Max)
  If Key$ = zCtrlY$ Or Key$ = zDel$ Or_
    Key$ = zBack$ Or Key$ = zCtrlT$ Then EditFlag = True
End Sub


Sub CharBack(E$, Max, PosAkt) Static
  If PosAkt > 1 Then
    E$ = Left$(E$, PosAkt – 2) + Mid$(E$, PosAkt) + " "
    PosAkt = PosAkt – 1
  End If
End Sub

Sub CharCtrlRight(E$, Max, PosAkt) Static
  While PosAkt < Max
    PosAkt = PosAkt + 1
    If Instr(Terminator$, Mid$(E$, PosAkt - 1, 1)) <> Ø_
       And Instr(Terminator$, Mid$(E$, PosAkt, 1)) = Ø Then Exit Sub
  Wend
End Sub

Sub CharCtrlLeft(E$, Max, PosAkt) Static
  While PosAkt > 1
    PosAkt = PosAkt - 1
    If PosAkt = 1 Then Exit Sub
    If Instr(Terminator$, Mid$(E$, PosAkt - 1, 1)) <> Ø_
       And Instr(Terminator$, Mid$(E$, PosAkt, 1)) = Ø Then Exit Sub
  Wend
End Sub

Sub CharCtrlT(E$, Max, PosAkt) Static
  PosNeu = PosAkt
  Call CharCtrlRight(E$, (Max), PosNeu)
  E$ = Left$(E$, PosAkt - 1) + Mid$(E$, PosNeu)
  Call Fill(E$, Max)
End Sub

'-------------------------------------------------------------
```

```
'       Line-Input-Routine
'       ------------------
'       Funktion: Eingabe in definierter Zone
'       Aufruf  : Call LInput(E$, (Col), (Row), (Max),(Char$), (Mode),_
'                             (Back$), Insert, PosAkt, Last$, EditFlag)
'       Hin     : E$       : Eingabevorgabe
'                 Col/Row  : Start Eingabezone
'                 Max      : Maximale Zeichenanzahl
'                 Char$    : Zulässige Zeichen
'                 Mode     : Ø=StandardColors/1=invers
'                 Back$    : Endezeichen
'                 Insert   : Ø=Einfügen/1=Überschreiben
'                 PosAkt   : Startposition des Cursors in der Zone
'       Zurück  : E$       : Eingabe
'                 Insert   : False=Überschreiben/True=Einfügen
'                 PosAkt   : Letzte Cursorposition in der Zone
'                 Last$    : Endezeichen
'                 EditFlag : False=keine Änderungen/True=Änderungen


Sub LInput(E$, Col, Row, Max, Char$, Mode,_
           Back$, Insert, PosAkt, Key$, EditFlag) Static

'       * Initialisierung *
  Call SetCol((Mode))
  Call Fill(E$,(Max))
  EditFlag = False
  Key$ = " "

'       * Eingabe *
  While Instr(Back$, Key$) = Ø
    Locate Row, Col, Ø: Print E$; :Locate Row, Col + PosAkt - 1, 1
    Call Taste(Key$, Escape)
    If Instr(Char$, Key$) <> Ø And Escape = False Then
      Call CharInput(E$, (Key$), (Max), PosAkt, (Insert))
      EditFlag = True
    Else
      Call EditKey((Key$), E$, (Max), PosAkt, EditFlag, Insert)
    End If
  Wend
  Call Compress(E$)
  Call SetCol(True)
  Locate ,,Ø
End Sub
```

5.3 Einbindung in die User-Library

Die Eingaberoutine werden wir noch sehr oft benötigen. Um das immer wiederkehrende »Mitkompilieren« zu vermeiden, kompilieren wir die Routine als eigenständiges Objektmodul, das wir anschließend zusätzlich zu ROUTINEN in die User-Library einbinden.

Wenn Sie es nicht bereits taten, kopieren Sie nun INPUT.BAS von der beiliegenden Diskette auf Ihre Arbeitsdiskette (Festplattenbesitzer: in das Verzeichnis QB).

Die Problematik der globalen Variablen im Zusammenhang mit eigenständigen Objektmodulen ist bereits berücksichtigt. Am Anfang von INPUT.BAS befindet sich die Anweisung $INCLUDE: 'comdef.bas'. Bei der Kompilierung wird die Deklarationsdatei eingebunden. Das Eingabemodul kennt nun die benötigten globalen Variablen.

Die Einbindung ist dank der erläuterten Batch-Dateien kinderleicht. Verlassen Sie bitte QuickBASIC. Rufen Sie zuerst den »Kompilier-Batch« auf.

Diskette : `DC INPUT`
Festplatte : `PC INPUT`

INPUT.BAS wird kompiliert und die erzeugte Objektdatei INPUT.OBJ auf die Programmdiskette kopiert (Festplatte: in das Verzeichnis SYSTEM20). Und nun rufen Sie den »Library-Batch« auf.

Diskette : `DL ROUTINEN INPUT`
Festplatte : `PL ROUTINEN INPUT`

Auf der Arbeitsdiskette (Festplatte: in QB) wird eine User-Library erstellt, die die beiden Module ROUTINEN.OBJ und INPUT.OBJ enthält. Das war alles! Wenn Sie QuickBASIC mit *QB/L* aufrufen, wird die erweiterte Library resident geladen.

5.4 Aufruf der Eingaberoutine (INPUTDEM.BAS)

Laden Sie bitte QuickBASIC mit *QB/L*. Die User-Library – und damit sowohl die Routinen-Sammlung als auch die Eingaberoutine – wird resident geladen.

Probieren wir die Eingaberoutine mit einem kleinen Demoprogramm aus. Es initialisiert die globalen Variablen und ruft LINPUT auf.

```
'       Demo der INPUT-Routine
'       ----------------------

REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
'-----------------------------------------------------------------------
```

```
'       Mainprog
'       --------
E$    = "Testeingabe"
Mode  = False: Insert = True: PosAkt = 1

Call lInput(E$, 5, 10, 20, (Alpha$), (False),_
            zEsc$ + zReturn$, Insert, PosAkt, Last$, EditFlag)
Print: Print "Eingabestring: " E$
```

Dieses Demoprogramm befindet sich wie alle Demoprogramme im Verzeichnis DEMO, und zwar unter dem Namen INPUTDEM.BAS. Laden Sie INPUTDEM.BAS, kompilieren Sie das Programm im Speicher und führen Sie es aus.

Durch Einbindung von COMDEF.BAS werden die globalen Variablen deklariert und anschließend durch Einbindung von INIT.BAS initialisiert. Danach werden die Parameter für die Demonstrationseingabe festgelegt und die Eingaberoutine aufgerufen.

Sie können die Vorgabe »Testeingabe« nun nach Belieben editieren. Als Endetaste wurde <zReturn$> definiert, also die RETURN-Taste.

Nach Beenden der Eingabe mit RETURN ruft das Hauptprogramm die Prozedur COMPRESS (in der Routinen-Sammlung) auf, die alle überflüssigen Leerzeichen entfernt und <E$> auf die echte Länge komprimiert. Anschließend wird zur Kontrolle die Eingabe selbst, die Länge der Eingabe, der Zustand des »Einfügen-/Überschreib-Flags« und der Zustand des »Änderungs-Flags« ausgegeben.

Um Ihnen die Anwendung von LINPUT zu erleichtern, fasse ich die übergebenen Parameter noch einmal kurz zusammen.

Hin:

- <E$>: Vorgabe
- <Col>/<Row>/<Max>: Startposition und Länge der Eingabezone
- <Char$>/<Back$>: Zulässige Eingabe- beziehungsweise Endezeichen
- <Mode>/<Insert>: Flag für Darstellungsmodus (normal/invers)/ Flag für Eingabemodus
- <PosAkt>: Startposition des Cursors in der Eingabezone

Zurück:

- <E$>: Eingabe
- <Insert>/<EditFlag>: Aktueller Eingabemodus nach Verlassen der Eingaberoutine / Flag, das anzeigt, ob die Vorgabe verändert wurde

- <PosAkt>: Letzte Position des Cursors in der Eingabezone vor dem Beenden der Eingabe
- <Last$>: Endezeichen

Alle Parameter, die von LINPUT zurückübergeben werden, müssen selbstverständlich nach Referenz übergeben werden! Denken Sie daran: Bei allen anderen – per Wert übergebenen – Variablen haben Sie die Wahl. Sie können sie als Variablen, als Konstanten oder gar als gemischte Ausdrücke übergeben (siehe Demo: Der Ausdruck (!) *zEsc$ + zReturn$* definiert die Endetasten).

Warum <Insert>, <Last$> und <PosAkt> an das aufrufende Programm zurückübergeben werden, habe ich noch nicht erläutert. Die Rückübergabe dieser Parameter wird zwar nicht für einzelne Eingaben, aber für die Verwaltung von Masken benötigt.

<Last$> kann bei der Eingabe in eine Maske außer <zReturn$> auch eine Cursortaste sein (<zDown$> oder <zUp$>). Das aufrufende Programm – die »Maskensteuerung« – muß unbedingt wissen, mit welcher Taste die Eingabe beendet wurde, um entsprechend reagieren zu können. Zum Beispiel könnte RETURN die gesamte Maskeneingabe beenden, und mit den Cursortasten OBEN/UNTEN könnte der Cursor zum nächsten/vorhergehenden Feld der Maske bewegt werden.

LINPUT muß auch den aktuellen Zustand des Flags <Insert> zurückübergeben können. Sonst passiert folgendes: LINPUT wird zum ersten Mal aufgerufen, <Insert> wurde im Hauptprogramm mit <True> initialisiert (=Einfügemodus eingeschaltet). Der Benutzer schaltet mit der Taste INS in den Überschreibmodus und editiert das erste Feld der Maske.

Nun bewegt er mit CURSOR UNTEN den Cursor zum nächsten Feld. LINPUT wird verlassen, die Maskensteuerung ruft LINPUT erneut auf, diesmal mit den Parametern für das nächste Feld (dessen Startposition, Länge etc.). Und nun wundert sich der Benutzer, warum plötzlich wieder der Einfügemodus eingeschaltet ist! Weil LINPUT die Veränderung des Flags <Insert> dem aufrufenden Programm nicht »mitteilen« konnte. LINPUT kann bei Übergabe nach Wert die Hauptprogramm-Variable <Insert> nicht verändern.

Um solche Überraschungen zu vermeiden, muß <Insert> per Referenz übergeben werden. LINPUT kann nun den letzten Zustand dieses Flags an das aufrufende Programm zurückübergeben.

<PosAkt> könnte theoretisch sowohl bei Einzeleingaben als auch bei der Maskenverwaltung von LINPUT immer mit 1 initialisiert werden. Warum sollte es sinnvoll sein, daß sich der Cursor nach dem Aufruf von LINPUT

statt auf dem ersten auf dem zweiten oder dritten Zeichen der Eingabezone befindet?

Nun, die Eingaberoutine kann auch zum Aufbau eines Bildschirm-Editors verwendet werden. Mit einem Bildschirm-Editor arbeiten alle Programm-Editoren, QuickBASIC, Turbo Pascal und so weiter.

Ein Bildschirm-Editor kann als eine spezielle Art der Eingabemaske aufgefaßt werden. Jede Bildschirmzeile ist eine 80 Zeichen lange Eingabezone. Die erste Eingabezone beginnt ab Spalte 1 von Zeile 1, die zweite Zone ab Spalte 1 von Zeile 2 und so weiter. Wie in einer Maske kann sich der Benutzer mit CURSOR UNTEN/OBEN von Feld zu Feld bewegen.

Im Gegensatz zu »normalen« Masken darf der Cursor jedoch keinesfalls immer auf den Anfang der Eingabezone gesetzt werden. Was würden Sie davon halten, wenn Sie im QuickBASIC-Editor in der Mitte einer Zeile sind, den Cursor nach unten bewegen und sich nun am Anfang der nächsten Zeile wiederfinden?

Angenommen, der Benutzer bewegt den Cursor mit CURSOR UNTEN zur nächsten Zeile. Die Eingaberoutine übergibt dem aufrufenden Bildschirm-Editor in <PosAkt> die letzte Cursorposition in der aktuellen Zeile.

Der Bildschirm-Editor ruft LINPUT erneut auf, mit den Parametern der neuen Eingabezone (=Bildschirmzeile). <PosAkt> wird unverändert übergeben. Das heißt, die letzte Cursorposition ist zugleich die Startposition in der neuen Eingabezone. Der Cursor befindet sich in der nächstunteren Zeile auf der gleichen Spalte wie zuletzt in der darüberliegenden Zeile.

6

Maskensteuerung

Im letzten Kapitel behauptete ich, mit LINPUT ließen sich Bildschirmmasken verwalten oder gar Bildschirm-Editoren erstellen. Daß diese Behauptung nicht übertrieben ist, beweist die Prozedur MASKE.

MASKE basiert auf der Eingaberoutine, die inzwischen in unsere User-Library integriert ist. Die Eingaberoutine LINPUT verwaltet einzelne Eingaben. Um daraus eine Maskensteuerung zu machen, benötigen wir eine Prozedur, die sich eine Ebene »über« LINPUT befindet.

Das Prinzip: Nach dem Aufruf ruft MASKE wiederum die Eingaberoutine auf und übergibt ihr die Parameter des ersten Feldes. LINPUT kontrolliert nun die Eingabe in dieses Feld. Nach Rückkehr aus LINPUT prüft MASKE das vom Benutzer verwendete Endezeichen und reagiert entsprechend.

Beispiel: Wurde die Eingabe mit CURSOR UNTEN (=Cursor zum nächsten Feld bewegen) beendet, ruft MASKE die Eingaberoutine erneut auf, übergibt diesmal jedoch die Parameter des zweiten Feldes. LINPUT überwacht nun die Eingabe in das zweite Feld der Eingabemaske.

MASKE soll folgende Anforderungen erfüllen:

1. Analog der Feldvorgabe bei der Eingaberoutine kann das aufrufende Programm MASKE einen »Maskeninhalt« vorgeben. Das aufrufende Programm kann MASKE für jedes einzelne Feld einen Inhalt übergeben. MASKE füllt diese Feldvorgaben auf die Maximallängen der einzelnen Felder auf und gibt sie auf dem Bildschirm aus – die Eingabemaske erscheint.

2. Wie in der Eingaberoutine kann der Benutzer jederzeit zwischen dem Einfüge- und dem Überschreibmodus umschalten. Das aufrufende Programm bestimmt, welcher der beiden Modi nach dem Aufruf der Maskensteuerung eingeschaltet ist.

3. Cursorsteuerung: CURSOR UNTEN bewegt den Cursor zum nächsten, CURSOR OBEN zum vorigen Feld. Zusätzlich sollen die Tastenkombinationen CTRL+HOME und CTRL+END den direkten »Sprung« zum ersten beziehungsweise letzten Feld ermöglichen (die Kombination mit CTRL wird benötigt, da die »einfachen« Tasten HOME und END

bereits von der Eingaberoutine zum Sprung an den Feldanfang beziehungsweise an das Feldende verwendet werden).

4. Das aufrufende Programm übergibt MASKE einen String mit jenen Tasten, die die komplette Maskeneingabe beenden sollen (zum Beispiel ESC oder RETURN). MASKE wiederum übergibt dem aufrufenden Programm nach dem Beenden der Eingabe das vom Anwender benutzte Endezeichen. Zusätzlich übergibt MASKE ein Flag, das anzeigt, ob der Benutzer eine der Vorgaben – also den Inhalt der Maske – veränderte.

MASKE benötigt die Parameter aller Felder der Eingabemaske, die Feldpositionen, Feldlängen und so weiter. Diese Parameter werden in eindimensionalen Arrays übergeben. Der Aufruf lautet:

```
Sub Maske(E$(1), Col(1), Row(1), Max(1), Char$(1),_
          Mode(1), Back$, Insert, Last$, MaskEdit) Static
```

Parameter hin:

- <E$(..)>: vorgegebene Feldinhalte
- <Col(..)>/<Row(..)>/<Max(..)>: Startpositionen der einzelnen Felder und Feldlängen
- <Char$(..)>: die in den einzelnen Feldern zulässigen Eingabezeichen
- <Mode(..)>: Feld-Darstellungsmodus (Mode=True => normale Darstellung; Mode=False => inverse Darstellung)
- <Back$>: String mit jenen Tasten, die die Eingabe in die Maske beenden
- <Insert>: Flag, das angibt, ob beim Aufruf der Maske der Einfüge- oder der Überschreibmodus eingeschaltet sein soll (True=Einfügen; False=Überschreiben).

Parameter zurück:

- <E$(..)>: Nach Rückkehr aus der Prozedur MASKE enthält das Array <E$(..)> die Inhalte der einzelnen Felder.
- <Insert>: Flag, das aussagt, ob zuletzt der Einfüge- oder der Überschreibmodus eingeschaltet war (True=Einfügen; False=Überschreiben).
- <Last$>: Das Zeichen, mit dem die Maskeneingabe beendet wurde.
- <MaskEdit>: Flag, das aussagt, ob die vorgegebenen Feldinhalte verändert wurden (MaskEdit=True => Änderungen vorgenommen; MaskEdit=False => keine Änderungen).

Das folgende Beispiel zeigt, wie mit MASKE eine Eingabemaske verwaltet wird, die aus zwei Feldern besteht:

```
Dim E$(2), Col(2), Row(2), Max(2), Char$(2), Mode(2)

E$(1)    = "Maier"     : E$(2)    = "Hans"        'Feldvorgaben
Col(1)   = 5: Row(1)=2: Col(2)   = 30: Row(2)=2 'Startpositionen
Max(1)   = 20          : Max(2)   = 10            'Feldlängen
Char$(1)= Alpha$       : Char$(2)= Alpha$         'Eingabezeichen
Mode(1) = False        : Mode(2) = True           'Darstellungsmodus

Back$    = zEsc$ + zReturn$  'Tasten, die die Maskeneingabe beenden
Insert   = True              'Einfügemodus einschalten

Call Maske(E$(), Col(), Row(), Max(), Char$(),_
           Mode(), Back$, Last$, MaskEdit)
```

MASKE verwaltet zwei Felder, die bei Spalte 5 von Zeile 2 beziehungsweise bei Spalte 30 von Zeile 2 beginnen. Feld 1 besitzt eine Länge von 20, Feld 2 von 10 Zeichen.

```
    ^     (Feld 1)     ^     ^(Feld 2)^

...
...
...
```

6.1 Programmentwicklung

Am Anfang der Prozedur wird wie immer die Datei COMDEF.BAS mit der $INCLUDE-Anweisung eingebunden. Dann folgt der übliche Kommentarblock mit einer Zusammenfassung der Parameterübergabe.

```
'       ********************
'       *  Maskensteuerung  *
'       ********************

'       *** (C) Said Baloui, 1987 ***

REM $INCLUDE: 'comdef.bas'
'-----------------------------------------------------------------------

'       Maskensteuerung
'       ---------------
'       Funktion: Verwaltet Eingaben in eine Maske
'       Aufruf  : Call Maske(E$(), (Col()), (Row()), (Max()),_
'                           (Char$()), (Mode()), Back$,
'                           Last$, MaskEdit, Insert)
'       Hin     : E$()                : Vorgaben
'                 Col()/Row()/Max()   : Feldposition/Feldlänge
'                 Char$()             : zulässige Zeichen
'                 Mode()              : True=normal/False=invers
'                 Insert              : Eingabemodus (einfügen: ja/nein)
'       Zurück  : E$()                : Eingaben
'                 Back$               : Endezeichen für Maskeneingabe
'                 Insert              : Eingabemodus (einfügen: ja/nein)
'                 Last$               : Benutztes Endezeichen
'                 MaskEdit            : False=keine/True=Änderungen

Sub Maske(E$(1), Col(1), Row(1), Max(1), Char$(1),_
          Mode(1), Back$, Insert, Last$, MaskEdit) Static
```

Ausgabe der Maske

Im Array <E$(..)> übergibt das aufrufende Programm einen vorgegebenen Maskeninhalt. Vor Beginn der Editierung gibt MASKE diese Feldvorgaben auf dem Bildschirm aus.

```
'       * Maske ausgeben *
  For i=LBound(E$) To UBound(E$)
    Call SetCol((Mode(i)))        'Darstellungsart f.betr.Feld einschalten
    Locate Row(i), Col(i), Ø      'Cursor auf Zonenanfang setzen
    Call Fill(E$(i),(Max(i)))     'Feldinhalt auf Maximallänge auffüllen
    Print E$(i)                   'Feldvorgabe ausgeben
  Next i
```

Der Start- und der Endwert der Ausgabeschleife wird »dynamisch« ermittelt. *LBOUND(E$)* ermittelt die untere Arraygrenze, also den niedrigsten Index des Arrays <E$(..)>. *UBOUND(E$)* ermittelt die obere Arraygrenze, den Index des letzten Feldes der Maske. Der Ablauf bei der Ausgabe eines Feldes:

- Mit der Prozedur SETCOL (in der Eingaberoutine enthalten) wird der übergebene Darstellungsmodus eingeschaltet (Mode(i)=True => normal; Mode(i)=False => invers).
- Der Cursor wird auf die Startposition des Feldes gesetzt.

- Mit FILL (in der Routinen-Sammlung enthalten) wird die Feldvorgabe mit Leerzeichen auf die Maximallänge <Max(i)> aufgefüllt und danach ausgegeben.

Variablen-Initialisierung

Bevor es »losgeht«, sind noch einige Variablen zu initialisieren.

```
'        * Variablen-Initialisierung *
  Ende$     = Back$ + zDown$ + zUp$ +_
              zCtrlHome$ + zCtrlEnd$ + zCtrlY$
  Nr        = LBound(E$)          'Start mit erstem Arrayelement (=1.Feld)
  MaskEdit  = False               '(noch) keine Änderungen
  Last$     = " "                 'Taste wegen WHILE initialisieren
```

<Ende$> enthält die Endezeichen für die Eingaberoutine LINPUT, also jene Tasten, mit denen eine einzelne und von LINPUT kontrollierte Eingabe beendet und zu MASKE zurückgekehrt wird. Eine Eingabe wird beendet, wenn der Benutzer eine der in <Back$> enthaltenen Tasten drückt (in <Back$> übergibt das aufrufende Programm die Zeichen, die die gesamte Maskeneingabe beenden). Zusätzlich wird eine einzelne Eingabe beendet, wenn eine der bereits beschriebenen Tasten(-kombinationen) CURSOR UNTEN/OBEN/CTRL+HOME/CTRL+END/CTRL+Y gedrückt wird.

Die Variable <Nr> gibt an, welches Feld gerade editiert wird. <Nr> wird mit 1 initialisiert, der Nummer des ersten Feldes.

<MaskEdit>, das »Änderungs-Flag«, wird mit <False> initialisiert. Enthält es nach Rückkehr aus MASKE den gleichen Wert, wurden die Feldvorgaben nicht verändert. Daß <LAST$> (=Zeichen, mit dem die Eingabe beendet wurde) ein Leerzeichen zugewiesen wird, klärt sich gleich von selbst.

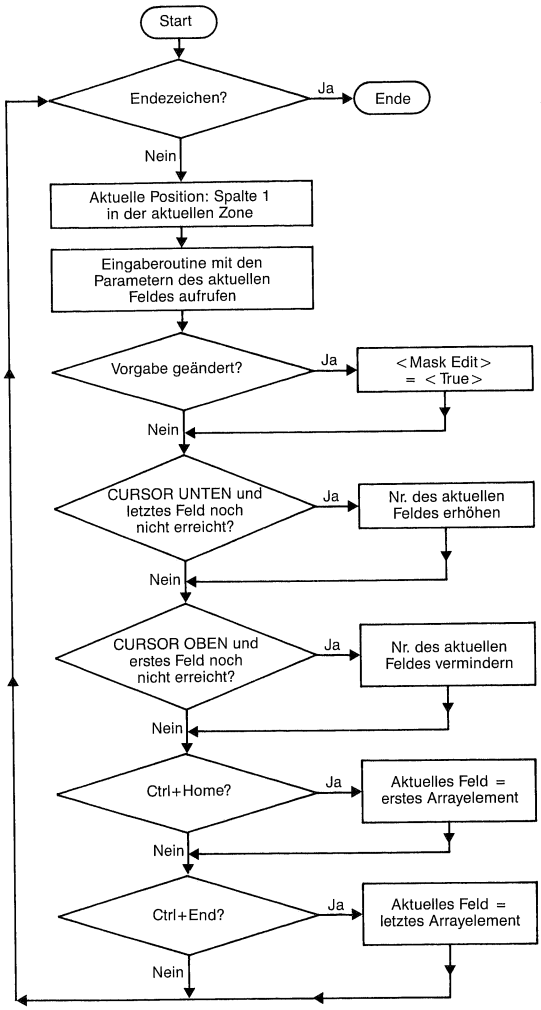


Bild 5 *Editierschleife*

Die nun folgende Schleife steuert die Editierung der Maske. In der Schleife wird ermittelt, welches Feld der Benutzer editieren will, und LINPUT wird mit den Parametern dieses Feldes aufgerufen.

Es handelt sich um eine WHILE-Schleife, die erst verlassen wird, wenn LINPUT ein Endezeichen übergibt, das in <Back$> enthalten ist (<Back$> = Zeichen, die die Maskeneditierung beenden).

```
'        * Editierschleife *
  While Instr(Back$, Last$) = Ø
    PosAkt   = 1
    Call lInput(E$(Nr), (Col(Nr)), (Row(Nr)), (Max(Nr)), (Char$(Nr)),_
                (Mode(Nr)), (Ende$), Insert, PosAkt, Last$, EditFlag)
    If EditFlag = True Then MaskEdit = True        'Maske geändert
    If Last$ = zDown$ And Nr < UBound(E$) Then Nr = Nr + 1
    If Last$ = zUp$   And Nr > LBound(E$) Then Nr = Nr - 1
    If Last$ = zCtrlHome$ Then Nr = LBound(E$)
    If Last$ = zCtrlEnd$  Then Nr = UBound(E$)
  Wend
End Sub
```

<Last$> wurde mit einem Leerzeichen initialisiert, da die Schleife sonst nicht einmal durchlaufen wird. Enthält <Last$> nichts (*Last$=" "*), übergibt die INSTR-Funktion einen Wert ungleich null und die Bedingung *While Instr(Back$, Last$) = 0* ist nicht erfüllt.

Dieses auf den ersten Blick »blödsinnige« Verhalten von INSTR besitzt eine gewisse Logik. Man könnte sagen: »nichts« ist immer in einem String enthalten. Doch nun zu den Vorgängen innerhalb der Schleife:

- LINPUT wird mit den Parametern des aktuellen Feldes <Nr> aufgerufen. Vor dem Aufruf wird <PosAkt> – die aktuelle Cursorposition – auf eins gesetzt.

- Wenn nach der Rückkehr aus LINPUT <EditFlag> den Wert <True> besitzt, wurde der Inhalt des aktuellen Feldes geändert. <MaskEdit> wird auf <True> gesetzt, um dem aufrufenden Programm später mitzuteilen, daß der Benutzer die vorgegebenen Maskeninhalte änderte.

- Behandlung der Cursortasten: CURSOR UNTEN bewegt den Cursor zum nächsten Feld. Wenn nicht bereits das letzte Feld editiert wird (wenn die Bedingung *Nr < UBound(E$)* erfüllt ist), wird <Nr> um eins erhöht.

  Entsprechend wird <Nr> um eins verringert, wenn der Benutzer die Eingabe mit CURSOR OBEN beendete und <Nr> größer als die Array-Untergrenze ist.

  CTRL+HOME setzt den Cursor auf das erste Feld (*LBound(E$)*), CTRL+END auf das letzte Feld (*UBound(E$)*).

6.2 Listing der Maskensteuerung (MASKE.BAS)

```
'       ********************
'       *  Maskensteuerung  *
'       ********************

'       *** (C) Said Baloui, 1987 ***

REM $INCLUDE: 'comdef.bas'
'-----------------------------------------------------------------------------

'       Maskensteuerung
'       ---------------
'       Funktion: Verwaltet Eingaben in eine Maske
'       Aufruf  : Call Maske(E$(), (Col()), (Row()), (Max()),_
'                           (Char$()), (Mode()), Back$,
'                           Last$, MaskEdit, Insert)
'       Hin     : E$()                : Vorgaben
'                 Col()/Row()/Max()   : Feldposition/Feldlänge
'                 Char$()             : zulässige Zeichen
'                 Mode()              : True=normal/False=invers
'                 Insert              : Eingabemodus (einfügen: ja/nein)
'       Zurück  : E$()                : Eingaben
'                 Back$               : Endezeichen für Maskeneingabe
'                 Insert              : Eingabemodus (einfügen: ja/nein)
'                 Last$               : Benutztes Endezeichen
'                 MaskEdit            : False=keine/True=Änderungen

Sub Maske(E$(1), Col(1), Row(1), Max(1), Char$(1),_
          Mode(1), Back$, Insert, Last$, MaskEdit) Static

'       * Maske ausgeben *
  For i=LBound(E$) To UBound(E$)
    Call SetCol((Mode(i)))        'Darstellungsart f.betr.Feld einschalten
    Locate Row(i), Col(i), Ø      'Cursor auf Zonenanfang setzen
    Call Fill(E$(i),(Max(i)))     'Feldinhalt auf Maximallänge auffüllen
    Print E$(i)                   'Feldvorgabe ausgeben
  Next i

'       * Variablen-Initialisierung *
  Ende$    = Back$ + zDown$ + zUp$ +_
             zCtrlHome$ + zCtrlEnd$ + zCtrlY$
  Nr       = LBound(E$)           'Start mit erstem Arrayelement (=1.Feld)
  MaskEdit = False                '(noch) keine Änderungen
  Last$    = " "                  'Taste wegen WHILE initialisieren

'       * Editierschleife *
  While Instr(Back$, Last$) = Ø
    PosAkt   = 1
    Call lInput(E$(Nr), (Col(Nr)), (Row(Nr)), (Max(Nr)), (Char$(Nr)),_
                (Mode(Nr)), (Ende$), Insert, PosAkt, Last$, EditFlag)
```

```
    If EditFlag = True Then MaskEdit = True        'Maske geändert
    If Last$ = zDown$ And Nr < UBound(E$) Then Nr = Nr + 1
    If Last$ = zUp$   And Nr > LBound(E$) Then Nr = Nr - 1
    If Last$ = zCtrlHome$ Then Nr = LBound(E$)
    If Last$ = zCtrlEnd$  Then Nr = UBound(E$)
  Wend
End Sub
```

6.3 Einbindung in die User-Library

MASKE.BAS ist ein eigenständiges Modul, das wir in die User-Library einbinden werden. Wie üblich beginnt das Programm mit $INCLUDE: 'comdef.bas', um die Deklarationsdatei einzubinden. Kopieren Sie bitte MASKE.BAS von der Diskette zum Buch auf Ihre Arbeitsdiskette und geben Sie ein:

Diskette : `DC MASKE`
Festplatte : `PC MASKE`

MASKE wird kompiliert und das erzeugte Objektmodul MASKE.OBJ auf der Programmdiskette (Laufwerk B) beziehungsweise im Verzeichnis SYSTEM20 gespeichert.

Dort befinden sich – wenn Sie inzwischen keine Löschaktionen durchführten! – immer noch die Module ROUTINEN.OBJ und INPUT.OBJ. Alle drei Module binden wir nun in eine neue User-Library ein.

Diskette : `DL ROUTINEN INPUT MASKE`
Festplatte : `PL ROUTINEN INPUT MASKE`

Wir kompilieren nur MASKE.BAS! Alle weiteren benötigten Objektmodule sind bereits vorhanden. Es ist also keinesfalls nötig, immer alle Programme zu kompilieren, die eine User-Library bilden sollen. Kompilieren Sie nur neue oder geänderte Programme.

6.4 Aufruf der Maskensteuerung (MASKDEMO.BAS)

Laden Sie bitte QuickBASIC zusammen mit der neuen User-Library. Laden Sie anschließend von der Diskette zum Buch (Verzeichnis DEMOS) die Datei MASKDEMO.BAS, ein Demoprogramm, das die Anwendung der Maskensteuerung zeigt.

```
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
'-----------------------------------------------------------------------------

'       Mainprog
'       --------
Felder = 9
Dim E$(Felder), Col(Felder), Row(Felder),_
    Max(Felder), Char$(Felder), Mode(Felder)

E$(1)="Maier"  :Col(1)=5:Row(1)=1:Max(1)=20:Char$(1)=Alpha$:Mode(1)=False
E$(2)="Werner" :Col(2)=5:Row(2)=2:Max(2)=15:Char$(2)=Alpha$:Mode(2)=True
E$(3)="Aalweg2":Col(3)=5:Row(3)=3:Max(3)=20:Char$(3)=Alpha$:Mode(3)=True
E$(4)="8000"   :Col(4)=5:Row(4)=4:Max(4)=4 :Char$(4)=Num$  :Mode(4)=True
E$(5)="München":Col(5)=5:Row(5)=5:Max(5)=20:Char$(5)=Alpha$:Mode(5)=True
E$(6)="12345"  :Col(6)=5:Row(6)=6:Max(6)=15:Char$(6)=Alpha$:Mode(6)=True
E$(7)="5.11.59":Col(7)=5:Row(7)=7:Max(7)=8 :Char$(7)=Alpha$:Mode(7)=False
E$(8)="ledig"  :Col(8)=5:Row(8)=8:Max(8)=10:Char$(8)=Alpha$:Mode(8)=True
E$(9)="Student":Col(9)=5:Row(9)=9:Max(9)=15:Char$(9)=Alpha$:Mode(9)=False

Back$  = zEsc$ + zReturn$        'Beenden mit ESC und RETURN
Insert = True                    'Einfügemodus einschalten
Call Maske(E$(), Col(), Row(), Max(), Char$(),_
           Mode(), Back$, Insert, Last$, MaskEdit)
Locate 15,1: For i=1 To Felder: Print E$(i): Next i
```

Den »Vorspann«, mit dem COMDEF.BAS und INIT.BAS eingebunden werden, sind Sie ja inzwischen gewohnt.

Nach dieser Einbindung wird eine Maske mit neun Feldern definiert, die an die Masken von Adreßverwaltungen angelehnt ist. Alle Felder beginnen ab Spalte 5. Jedes Feld befindet sich in einer eigenen Zeile. Die Felder 1 und 9 werden invers dargestellt (*Mode(1)=False / Mode(9)=False*). Im Feld »Postleitzahl« sind nur numerische Eingaben zulässig (*Char$(4)=Num$*), in allen anderen Feldern beliebige alphanumerische Zeichen.

Der Benutzer kann die Editierung der Maske mit ESC oder RETURN beenden (*Back$ = zEsc$ + zReturn$*). Vor dem Aufruf von MASKE wird der Einfügemodus eingeschaltet (*Insert = True*).

Denken Sie beim Ausprobieren bitte an die komfortablen Editiermöglichkeiten von MASKE, vor allem an

CTRL+HOME	:	Sprung zum ersten Feld
CTRL+END	:	Sprung zum letzten Feld

Zur Kontrolle gibt das Demoprogramm nach der Rückkehr aus MASKE alle Feldinhalte auf dem Bildschirm aus.

7

Windowing und Pull-down-Menüs

Mit einem letzten »Großprojekt« schließen wir das Thema User-Libraries ab. PULLDOWN ist ein umfangreiches Programm, das aus einer Vielzahl von Prozeduren besteht.

PULLDOWN erläutere ich nicht nur wegen der Komplexität des Programms extrem ausführlich. Dieses Programm »entwickeln« wir. Das heißt, ich werde Ihnen das Programm nicht »auf dem Präsentierteller überreichen«, sondern mit Ihnen Lösungen bei bestimmten Problemstellungen erarbeiten und die Vor- beziehungsweise Nachteile der verschiedenen Ansätze herausarbeiten.

Vielleicht ist es vermessen, aber ich habe den Ehrgeiz, Ihnen an diesem Programm vorzuführen, wie alle bisher besprochenen Eigenschaften von QuickBASIC möglichst elegant kombiniert werden. Elegant heißt, daß das Ziel ein trotz seiner Komplexität überschaubares Programm ist.

PULLDOWN soll eine Art Muster für die modulare Programmentwicklung darstellen, die QuickBASIC vor allem mit den Prozeduren stark unterstützt.

Die Module von PULLDOWN sind klar voneinander abgegrenzt und besitzen fest umrissene Aufgaben. Ich werde zuerst die Bedienung der erzeugten Pull-down-Menüs und die doch etwas komplexe Programmstruktur erläutern. Danach wissen Sie, in welche Module das Gesamtprogramm unterteilt ist. Diese Module entwickeln wir der Reihe nach. Den Abschluß bildet eine »Steuerungsprozedur«, die mit Hilfe der einzelnen Module komplette Pull-down-Menüs verwaltet.

Der Name PULLDOWN ist übrigens nicht ganz zutreffend. Sicher, mit diesem Programm können Sie Pull-down-Menüs verwalten. Aber das ist bei weitem nicht alles. PULLDOWN enthält alle Routinen, die zur Verwaltung beliebiger Windows benötigt werden. Es ist sogar möglich – gemäß den neuesten Modetrends bei Programmen –, mehrere Windows zu überlagern und einzeln wieder »wegzublenden«.

7.1 Schnelle Bildschirmausgabe mit PRINTF

Bevor wir uns an das eigentliche Programm wagen, stelle ich Ihnen eine kleine Assembler-Routine vor, die PULLDOWN recht häufig aufrufen wird. Diese Routine beschleunigt die Bildschirmausgabe enorm. Ohne diese Beschleunigung könnten wir unser Projekt vergessen. Die normale Ausgabe mit PRINT ist für die Verwaltung von Pull-down-Menüs einfach zu langsam. Nicht so sehr wegen der Ausgabe eines einzelnen Menüs, sondern aus einem anderen Grund.

Stellen Sie sich vor, Sie befinden sich in den QuickBASIC-Menüs und geben »Dauerfeuer« auf die Taste CURSOR RECHTS. Blitzschnell werden Menüs auf- und zugeklappt. Jedesmal wird dabei der komplette Menüinhalt ausgegeben.

Diese Geschwindigkeit ist mit dem PRINT-Befehl nicht zu erreichen. Die Ausgabe der Menüs wird der Cursorbetätigung »hinterherhinken«. Und diese Eigenschaft paßt nicht so ganz zu einer Routine, die »professionell« sein will.

Die einzige Möglichkeit, ein Nachhinken zu vermeiden, ist eine Assembler-Ausgaberoutine, die die Zeichen direkt in den Bildschirmspeicher schreibt. Die Routine nennt sich PRINTF. Auf der Diskette zum Buch befindet sich der Sourcecode (PRINTF.ASM) und die Objektdatei (PRINTF.OBJ). Welche Besonderheiten bei der Einbindung der Objektdatei in die User-Library zu beachten sind, klären wir später. Vorläufig interessiert uns nur die Anwendung von PRINTF.

Syntax: `Call PrintF(Spalte, Zeile, Zeichenkette$, Attribut)`

- <Col>/<Row>: Startposition der Ausgabe (vergleichbar dem LOCATE-Befehl)
- <Zeichenkette$>: auszugebende Zeichenkette
- <Attribut>: ein Wert, der die Art der Darstellung bestimmt (normal, invers, blinkend etc.)

Um Ihnen das Experimentieren mit verschiedenen Werten für <Attribut> zu ersparen, enthält COMDEF.BAS drei als global deklarierte »Attribut-Variablen«, <Normal>, <Invers> und <Blinkend>. In INIT.BAS werden diese Variablen mit den Werten für die betreffende Darstellungsart initialisiert.

```
Normal    = 7      'Attribut für Normaldarstellung
Invers    = 112    'Attribut für Inversdarstellung
Intensiv  = 15     'Attribut für intensive Darstellung
```

Wie vom PRINT-Befehl gewohnt, können Sie auch bei PRINTF wahlweise Konstanten oder Variablen als Argumente angeben. Die folgenden Beispiele sprechen für sich.

Beispiele:

1. Call PrintF(2, 10, "Test", Normal) = >
   Normale Ausgabe von »Test« ab Spalte 2 von Zeile 10
2. Call PrintF(1, 1, "Müller", Invers) = >
   Inverse Ausgabe von »Müller« ab Spalte 1 von Zeile 1
3. Col = 10: Row = 20: String$="Noch ein Test": Call PrintF(Col, Row, String$, Normal) = >
   Intensiv hervorgehobene Ausgabe von »Noch ein Test« ab Spalte 10 von Zeile 20

Bedienung der Pull-down-Menüs

Die beiden folgenden Abbildungen zeigen ein von PULLDOWN verwaltetes Menü. Pull-down-Menüs bestehen aus einer beliebigen, nur durch die Bildschirmgröße beschränkten Anzahl von Untermenüs, die beliebig auf- und zugeklappt werden.

```
   Datei  File  Export
 ┌──────────────────┐ Demo    der      Pull-Down-Menüs
 │ Eintragen        │ Demo    der      Pull-Down-Menüs
 │ Suchen/Editieren │ Demo    der      Pull-Down-Menüs
 ├──────────────────┤ Demo    der      Pull-Down-Menüs
 │ Ausgabefolge     │ Demo    der      Pull-Down-Menüs
 │ Reorganisation   │ Demo    der      Pull-Down-Menüs
 └──────────────────┘ Demo    der      Pull-Down-Menüs
Dies    ist   ein     Demo    der      Pull-Down-Menüs
Dies    ist   ein     Demo    der      Pull-Down-Menüs
Dies    ist   ein     Demo    der      Pull-Down-Menüs
Dies    ist   ein     Demo    der      Pull-Down-Menüs
Dies    ist   ein     Demo    der      Pull-Down-Menüs
Dies    ist   ein     Demo    der      Pull-Down-Menüs
Dies    ist   ein     Demo    der      Pull-Down-Menüs
```

Nach dem Aufruf von PULLDOWN ist das erste Menü »aktiv«. Mit den Tasten CURSOR RECHTS und CURSOR LINKS wird das aktive Menü zugeklappt und ein benachbartes Menü aktiviert.

```
   Datei  File  Export
Dies    ist  ┌──────────────────┐      Pull-Down-Menüs
Dies    ist  │ Satz             │      Pull-Down-Menüs
Dies    ist  │ Teil der Datei   │      Pull-Down-Menüs
Dies    ist  │ Komplette Datei  │      Pull-Down-Menüs
Dies    ist  ├──────────────────┤      Pull-Down-Menüs
Dies    ist  │ Druckparameter   │      Pull-Down-Menüs
Dies    ist  │ Codetabelle      │      Pull-Down-Menüs
Dies    ist  ├──────────────────┤      Pull-Down-Menüs
Dies    ist  │ Gerät            │      Pull-Down-Menüs
Dies    ist  └──────────────────┘      Pull-Down-Menüs
Dies    ist  ein      Demo    der      Pull-Down-Menüs
Dies    ist  ein      Demo    der      Pull-Down-Menüs
Dies    ist  ein      Demo    der      Pull-Down-Menüs
Dies    ist  ein      Demo    der      Pull-Down-Menüs
```

Außer dem Aktivieren mit den Cursortasten ist wie in QuickBASIC eine »Direktanwahl« möglich. Für die Direktanwahl wird die Tastenkombination SHIFT+Anfangsbuchstabe des Menünamens verwendet.

Ein bestimmtes Menükommando kann entweder mit CURSOR UNTEN/OBEN und RETURN oder durch den Anfangsbuchstaben des Kommandonamens (ohne SHIFT) selektiert werden.

Wie in den QuickBASIC-Menüs können innerhalb eines Menüs Kommandogruppen durch einen Strich voneinander getrennt werden. Beim Selektieren mit CURSOR UNTEN/OBEN wird dieser Trennstrich übergangen. Diese »Kleinigkeit« ist keineswegs so trivial, wie es scheint, sondern wird uns noch vor einige Probleme stellen.

- CURSOR RECHTS/LINKS: Aktivieren eines benachbarten Menüs
- SHIFT+Anfangsbuchstabe: Aktivierung des Menüs mit dem betreffenden Anfangsbuchstaben
- CURSOR UNTEN/OBEN: Selektieren eines Kommandos (Bestätigen mit RETURN)
- Anfangsbuchstabe des Kommandos: direkte Selektion eines Kommandos mit Bestätigung der Auswahl

7.2 Programmstruktur

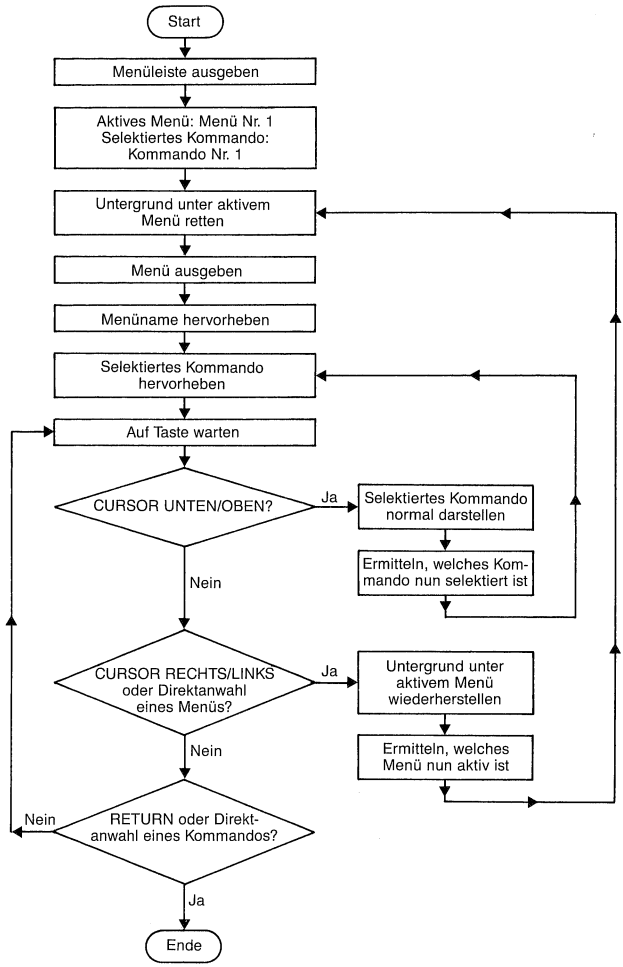


Bild 6 *Programmstruktur*

Im Gegensatz zu den vorhergehenden Projekten benötigt PULLDOWN einige Vorüberlegungen. Diesmal genügt es nicht, Ihnen den prinzipiellen Ablauf relativ grob zu erläutern und anschließend das Programm »häppchenweise« vorzustellen.

Vor der Umsetzung in BASIC-Befehle muß der Ablauf bis ins letzte Detail geklärt werden und zwar mit der üblichen Verfeinerungsmethode. Das heißt, zuerst erläutere ich die prinzipiellen Anforderungen und Abläufe anhand eines Übersichts-PAPs. Dieser Programmablaufplan wird schrittweise verfeinert, bis zuletzt mehrere »Fein-PAPs« vorliegen, die – relativ – problemlos in ein Programm umzusetzen sind.

Bild 6 zeigt den prinzipiellen Ablauf. Nach dem Aufruf von PULLDOWN wird die »Kopfzeile« oder »Menüleiste« ausgegeben. Sie enthält die Namen der Menüs.

Dann wird der Untergrund unter dem ersten Menü gerettet und das Menü »aufgeklappt«, also auf dem Bildschirm ausgegeben. Außerdem wird in der Kopfzeile der Name dieses Menüs und im Menü selbst das erste Kommando invers hervorgehoben.

Das Programm wartet auf eine Taste. Mit CURSOR UNTEN/OBEN wird ein benachbartes Kommando selektiert. Das invertierte Kommando wird normalisiert und das neu selektierte invertiert.

Angewählt wird ein Kommando über seinen Anfangsbuchstaben oder mit RETURN (Anwählen des momentan selektierten Kommandos). Das aktive Menü wird zugeklappt (der gerettete Untergrund auf den Bildschirm zurückgeschrieben). Dann wird dem aufrufenden Programm die Nummer des aktiven Menüs und des gewählten Kommandos übergeben.

CURSOR UNTEN/OBEN und die Direktanwahl mit SHIFT+Anfangsbuchstaben aktivieren ein neues Menü. Zuerst wird das momentan aktive Menü inaktiviert, indem der alte Untergrund wiederhergestellt und der Menüname in der Kopfzeile wieder normal dargestellt wird. Anschließend wird ermittelt, welches Menü nun aktiv ist, der Untergrund dieses Menüs gerettet und das Menü ausgegeben. Zusätzlich wird der betreffende Menüname in der Kopfzeile und das erste Kommando invertiert.

Eine Bitte: »Klammern« Sie sich nicht an diesen Ablauf und den zugehörigen PAP. Diese Darstellung ist noch viel zu grob und wird dem Programm in keiner Weise gerecht. Es geht momentan nur darum, einen Überblick über die Programmstruktur zu gewinnen.

7.3 Definition der Menüs

Bevor wir uns an Einzelheiten wagen, müssen wir klären, wie ein Pull-down-Menü überhaupt definiert wird, in welcher Weise das aufrufende Programm den Inhalt der Menüs festlegt. Um uns darüber klar zu werden, verwenden wir als »Denkmodell« ein Pull-down-Menü mit zwei Untermenüs.

```
Editieren          Blockkommandos

Suchen             Markieren
Ersetzen           Löschen
                   Kopieren
                   Verschieben
```

Da jedes Menü eine Anzahl von Kommandos enthält, benötigen wir mindestens ein Stringarray. Wahrscheinlich ist Ihr erster Gedanke – zumindest war es mein erster –, jedes der beiden Menüs in einem eigenen Stringarray zu definieren.

Dann ist das Modul PULLDOWN jedoch nicht mehr sehr flexibel. Das Modul soll ja auch Pull-down-Menüs mit drei, vier oder noch mehr Menüs verwalten können. Das heißt, einmal sind im Aufruf als Parameter zwei, einmal drei und einmal vier Stringarrays zu übergeben.

```
CALL PULLDOWN(A$(), B$())
CALL PULLDOWN(A$(), B$(), C$())
CALL PULLDOWN(A$(), B$(), C$(), D$())
```

Da QuickBASIC nicht mitspielt, wenn wir eine unterschiedliche Parameteranzahl übergeben wollen, scheidet diese Lösung aus. Zwei Lösungen bieten sich an:

1. Ein mehrdimensionales Array. Jede Dimension entspricht einem Menü. Das Array muß entsprechend den größten vorstellbaren Pull-down-Menüs dimensioniert werden, zum Beispiel mit:

   ```
   Dim Menue$(10,10)
   ```

   Nun können Sie maximal zehn Menüs mit jeweils zehn Kommandos definieren.

2. Ein eindimensionales Array, in dem die einzelnen Menüs durch eine Art »Trennstring« voneinander unterschieden werden, zum Beispiel durch einen leeren String.

```
Dim Menue$(100)
Menue$(1)="Suchen"
Menue$(2)="Ersetzen"
Menue$(3)=""
Menue$(4)="Markieren"
Menue$(5)="Löschen"
Menue$(6)="Kopieren"
Menue$(7)="Verschieben"
```

Beide Lösungen besitzen den gleichen Nachteil: Um zehn Menüs mit jeweils bis zu zehn Kommandos zu verwalten, müssen Stringarrays mit 100 Strings angelegt werden – obwohl mit Sicherheit niemals alle 100 Strings für ein Pull-down-Menü benötigt werden!

Da QuickBASIC pro angelegtem String drei Byte benötigt, verschwendet diese Lösung kostbaren Speicherplatz. Zugegeben, nicht allzu viel, aber wenn man bei jedem Problem derart »großzügig« programmiert, summieren sich die kleinen Verschwendungen bald zu größeren Beträgen. Warum also nicht platzsparender programmieren, wenn möglich?

Es ist möglich, und zwar so: Pro Menü wird genau ein String verwendet. In diesem String sind die einzelnen Kommandos durch ein Sonderzeichen getrennt. Zur Verwaltung von bis zu zehn Menüs genügt mit dieser Methode ein Array mit zehn Strings.

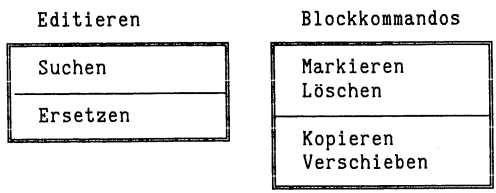
PULLDOWN erwartet als Trennzeichen ein »#« nach jedem Kommando. Unser Modellmenü wird daher so definiert:

```
Menue$(1)="Suchen#Ersetzen#"
Menue$(2)="Markieren#Löschen#Kopieren#Verschieben#"
```

Nun dürfen wir uns noch eine Methode zur Definition der Trennstriche zwischen den Kommandogruppen überlegen. Die einfachste Methode: Trennstriche zwischen Gruppen markieren wir ebenfalls mit einem Sonderzeichen, zum Beispiel mit einem Bindestrich.

```
Menue$(1)="Suchen#-#Ersetzen#"
Menue$(2)="Markieren#Löschen#-#Kopieren#Verschieben#"
```

Das Ergebnis dieser Menü-Definition:



Den Kommandos »Suchen« und »Markieren« folgt ein Bindestrich. PULLDOWN macht daraus Trennstriche innerhalb der beiden Menüs. Sie müssen zugeben: Kürzer und komfortabler kann die Menü-Definition kaum sein. Diesen Komfort müssen wir allerdings teuer bezahlen. PULLDOWN verwendet sehr komplexe Routinen zur Analyse des Menüarrays!

Außer den Menüstrings wird PULLDOWN die »Menüleiste« übergeben, die die einzelnen Menünamen enthält und eine Art »Überschrift« darstellt. Diese Leiste übergeben wir mit einem String. Die Positionen, an denen sich die Menünamen innerhalb des Strings befinden, bestimmen zugleich die Positionen der zugehörigen Menüs.

In unserem Beispiel beginnen die Kommandos des ersten Menüs »Editieren« ab Spalte 4 und die Kommandos des Menüs »Blockkommandos« ab Spalte 22. Die zugehörige Menüleiste:

```
Header$="   Editieren          Blockkommandos
"
```

Der »Leistenstring« sollte immer 80 Zeichen lang sein. Er wird bei Aufruf von PULLDOWN invers ausgegeben, und es sieht nicht sehr schön aus, wenn nur ein Teil der obersten Bildschirmzeile invers dargestellt wird. Die Leiste befindet sich immer in Zeile 1. Die Menüs beginnen in der darunterliegenden Zeile.

Durch Veränderung des Leistenstrings können Sie die Menüs beliebig auf dem Bildschirm positionieren. Zwei Grenzfälle sind dabei zu beachten:

1. Schauen Sie sich bitte noch einmal die schematische Darstellung des Demomenüs an. Vor jedem Menükommando befinden sich genau drei Zeichen, zwei Leerzeichen und der Rahmen. Das heißt, der erste Menüname im Leistenstring darf erst ab dem vierten Zeichen beginnen, keinesfalls vorher!

2. Außerdem müssen Sie darauf achten, daß die Menüs nicht über den rechten Bildschirmrand »hinausragen«. Die Menüs sind so gestaltet, daß dem längsten Kommando zwei Leerzeichen und das Rahmenzeichen folgen. Achten Sie bei der Definition darauf, daß auch das letzte Menü noch komplett auf den Bildschirm paßt!

Die vollständige Definition des Demomenüs:

```
Header$="   Editieren          Blockkommandos
"
Menue$(1)="Suchen#-#Ersetzen#"
Menue$(2)="Markieren#Löschen#-#Kopieren#Verschieben#"
```

Der Aufruf von PULLDOWN ist geradezu trivial:

```
Call PullDown(Menue, Befehl)
```

Vor Aufruf von PULLDOWN definieren wir in den globalen Variablen <Header$> und <Menue$(..)> die Menüleiste und die Menüs. In den beiden beim Aufruf angegebenen Variablen <Menue> und <Befehl> wird uns PULLDOWN die Nummer des zuletzt aktiven Menüs und die Nummer des darin selektierten Kommandos übergeben. Kein Vergleich mit dem »Rattenschwanz« an Parametern beim Aufruf der Eingaberoutine oder der Maskensteuerung!

7.4 Verwaltungsmodule

Nun dürfen wir uns überlegen, welche »Grundbausteine« für das Programm benötigt werden, welche »Module«. Das komplexeste Modul ist die Prozedur MENUENEU. Sie übernimmt die Aufgabe, die Aktivierung eines neuen Menüs zu steuern (altes Menü zuklappen, neues aufklappen). Um diese Aufgabe zu erfüllen, werden verschiedene weitere Prozeduren benötigt:

1. PUSHSCREEN und PULLSCREEN zum Retten/Wiederherstellen eines Menüuntergrundes. »Retten« (PUSHSCREEN) bedeutet, daß die Zeichen vom Bildschirm gelesen und in ein Array kopiert werden. Beim »Zurückholen« (PULLSCREEN) wird der Arrayinhalt auf dem Bildschirm ausgegeben.
2. NAMENEU sorgt bei der Aktivierung eines neuen Menüs dafür, daß in der Kopfzeile der Name des bisher aktiven Menüs wieder normal dargestellt und dafür der Name des neuen aktiven Menüs invers hervorgehoben wird.
3. BEFEHLNEU kümmert sich um die Selektierung eines neuen Kommandos. Das bisher selektierte Kommando wird normalisiert und das neu selektierte Kommando hervorgehoben (nach Aktivierung eines neuen Menüs ist immer dessen erstes Kommando selektiert).
4. INITMENUE ermittelt die Parameter des zu aktivierenden Menüs, seine Position auf dem Bildschirm und die Kommandos, die es enthält.
5. PRINTMENUE gibt das aktive Menü auf dem Bildschirm aus (Rahmen und Inhalt). PRINTMENUE verwendet dafür die Prozedur RAHMEN, die ein Rechteck zeichnet.

NAMENEU und BEFEHLNEU benutzen zwei weitere Prozeduren. Für ihre Invertier-/Normalisieraufgaben müssen beide Prozeduren wissen, wel-

che Zeichen zu behandeln sind und wo sich diese auf dem Bildschirm befinden.

NAMENEU benutzt dafür die Prozedur GETNAME. GETNAME wird eine Menünummer übergeben. Die Prozedur ermittelt, wo sich innerhalb des Leistenstrings der zugehörige Menüname befindet und übergibt diesen Teil des Leistenstrings und die ermittelte Position.

NAMENEU kann diesen String nun an der betreffenden Stelle invers oder normal ausgeben. Die gesamte Menüleiste wird invers dargestellt. Soll NAMENEU die Hervorhebung eines Namens rückgängig machen, wird er daher invers ausgegeben, um sich nicht mehr vom Rest der Menüleiste abzuheben. Soll er dagegen hervorgehoben werden, gibt ihn NAMENEU in normaler Darstellung aus.

GETSTRING erfüllt eine ähnliche Aufgabe wie GETNAME. Diese Prozedur wird vor allem von BEFEHLNEU aufgerufen. Ihr wird die Nummer des aktiven Menüs und die Nummer eines gesuchten Kommandos in diesem Menü übergeben. GETNAME »fischt« den Namen aus dem zugehörigen Menüstring, damit ihn BEFEHLNEU anschließend in der gewünschten Darstellungsart ausgeben kann.

7.5 Programmentwicklung

Wir beginnen nun mit der eigentlichen Programmentwicklung. Ich gehe davon aus, daß Ihnen die Struktur inzwischen vertraut ist.

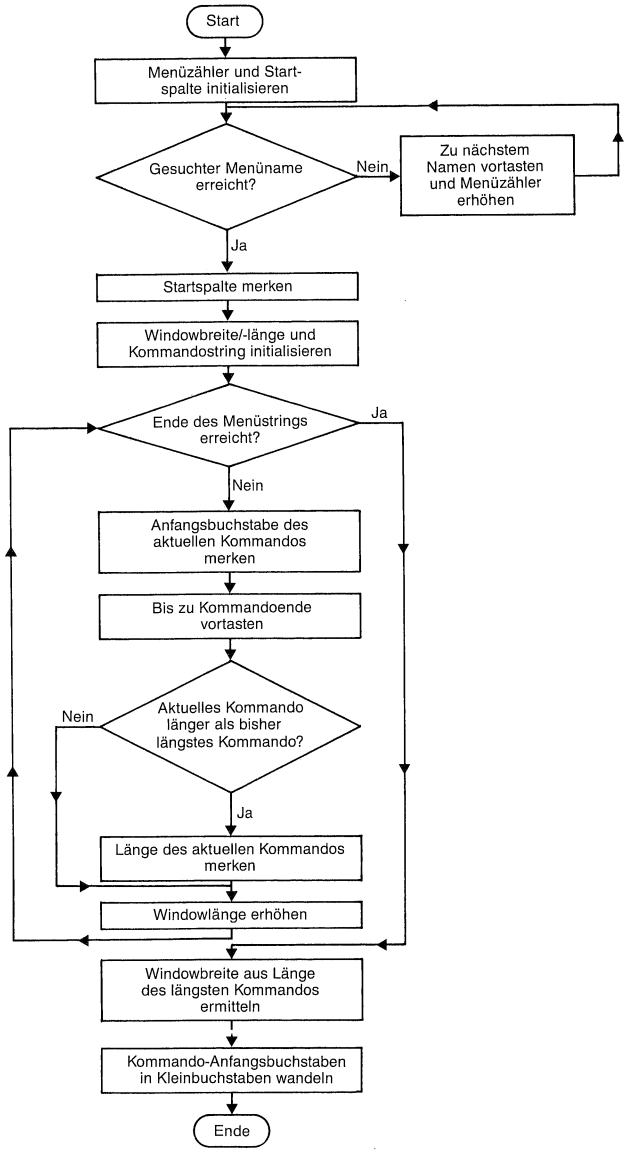


Bild 7 *Analyse eines Menüs (INITMENUE)*

Sie wissen nun, wie ein Pull-down-Menü definiert wird und welche Prozeduren wir zu seiner Verwaltung benötigen. Die grundlegendste Prozedur ist zweifellos INITMENUE. INITMENUE ermittelt die Parameter eines zu aktivierenden Menüs.

Aufgerufen wird INITMENUE unter Angabe der Menünummer. Die Prozedur ermittelt die Position dieses Menüs. Da jedes Menü ein Rechteck ist, ist die Position definiert durch:

1. Position der linken oberen Ecke (Spalte und Zeile). Die Zeile ist bereits »vordefiniert«. Alle Menüs beginnen unterhalb der Menüleiste in Zeile 2. Die Spalte hängt von der Position des zugehörigen Menünamens im Leistenstring ab. Alle Kommandos sollen in der gleichen Spalte wie der zugehörige Menüname beginnen.
2. Breite des Menüs (abhängig vom längsten Kommando innerhalb des Menüs
3. Länge des Menüs (abhängig von der Anzahl der Kommandos)

Außer der Position eines Menüs werden die Anfangsbuchstaben der darin enthaltenen Kommandos benötigt. Wie erläutert, können Kommandos direkt über diese Anfangsbuchstaben angewählt werden.

INITMENUE übergibt daher nicht nur die Menüposition, sondern zusätzlich einen String, der diese Anfangsbuchstaben enthält.

```
'       InitMenue
'       ---------
'       Funktion: Ermittelt Parameter von aktiviertem Menü
'       Hin     : Menue : Nr. des aktivierten Menüs
'       Zurück  : WCol/WBreite/WLaenge : Ermittelte Menüparameter

Sub InitMenue(Menue, WCol, WBreite, WLaenge, MBefehle$) Static
'       * Get Startspalte *
  Nr = Ø
  WCol = 1
  While Nr < Menue
    WCol = Instr(WCol, Header$, " ")
    Call SearchOther(Header$, " ", WCol)
    Nr = Nr + 1
  Wend
  WCol = WCol - WOffset
```

Zuerst wird die Startspalte des Menüs Nummer <Menue> ermittelt. INITMENUE tastet sich durch den Leistenstring, wobei <Nr> die aktuelle Menünummer ist.

Die aktuelle Menünummer <Nr> wird mit 0 initialisiert und der »Positionszeiger« <WCol> mit 1. Die folgende Schleife wird verlassen,

wenn <Nr> nicht mehr kleiner ist als <Menue>. Dann ist der gesuchte Menüname erreicht. Der Ablauf beim »Vortasten«:

1. Im Leistenstring wird ab der aktuellen Position <WCol> nach dem ersten Auftreten eines Leerzeichens gesucht (*WCol = Instr(WCol, Header$, " ")*. Diese Position befindet sich nun in <WCol>.
2. Da sich vor dem nächsten Menünamen beliebig viele Leerzeichen befinden können, wird SEARCHOTHER (in der Routinen-Sammlung enthalten) verwendet, um alle Leerzeichen zu überlesen. SEARCHOTHER mit dem Argument " " gibt die Position an, ab der sich kein Leerzeichen mehr befindet.
3. Wenn wir davon ausgehen, daß sich im Leistenstring nur Menünamen und Leerzeichen befinden, kennzeichnet <WCol> nun die Position des nächsten Menünamens. <WCol> »zeigt« auf seinen ersten Buchstaben. <Nr> wird entsprechend um eins erhöht.

Nach Verlassen der Schleife enthält <WCol> die Startspalte des betreffenden Menünamens. In der gleichen Spalte beginnen zwar auch die Kommandos, aber nicht das Menü selbst. Denken Sie an den Rahmen!

Wie weit links von den Kommandos der Rahmen beginnt, bestimmt die globale Variable <WOffset>.

<WOffset> wird in der Deklarationsdatei INIT.BAS der Wert 3 zugewiesen. Nach *WCol = WCol - WOffset* enthält <WCol> die Startspalte des Menünamens minus 3. Der Rahmen beginnt somit drei Zeichen links von den Kommandos und die Startspalte der Kommandos ist WCol + 3.

```
Blockkommandos
┌──────────────┐
│ Markieren    │
│ Löschen      │
├──────────────┤
│ Kopieren     │
│ Verschieben  │
└──────────────┘
```

Nun geht's um die Ermittlung der Breite und Länge des Menüs. Die Breite hängt vom längsten Kommando ab, die Länge von der Anzahl der Kommandos. Um beide Parameter zu ermitteln, muß die Prozedur den Menüstring <Menue$(Menue)> analysieren.

```
'       * Get Windowbreite/Windowlänge *
  WBreite   = Ø
  WLaenge   = 2
  MBefehle$ = ""
  j = Ø
  While j <> Len(Menue$(Menue))
    i = j + 1
    MBefehle$ = MBefehle$ + Mid$(Menue$(Menue), i, 1)
    j = Instr(i, Menue$(Menue), "#")
    If j - i > WBreite Then WBreite = j - i
    WLaenge = WLaenge + 1
  Wend
  WBreite = WBreite + 2 * WOffset
  Call UpLowCase(MBefehle$)
End Sub
```

<WBreite>, <WLaenge> und <MBefehle$> werden initialisiert. <WLaenge> wird nicht mit 0, sondern mit 2 initialisiert, um von vornherein die beiden Rahmenzeilen zu berücksichtigen. Zu diesen beiden Zeilen werden anschließend die Menüzeilen addiert. <MBefehle$> wird die Anfangsbuchstaben der Menükommandos enthalten.

Um die folgende Schleife zu verstehen, müssen Sie die Bedeutung der beiden »Zeiger« <i> und <j> kennen. Erinnern Sie sich, wie ein Menüstring aufgebaut ist: Kommando; Trennzeichen; Kommando; Trennzeichen und so weiter.

```
Menue$(2)="Markieren#Löschen#-#Kopieren#Verschieben#"
```

Die Prozedur tastet sich im Menüstring von Kommando zu Kommando vor. <j> zeigt immer auf das Ende des letzten Kommandos, also auf das Trennzeichen »#«. Das Kommandoende befindet sich genau ein Zeichen vor dem Beginn eines neuen Kommandos. <j> wird daher mit 0 initialisiert (=Beginn des ersten Kommandos minus 1).

Die WHILE-Schleife wird verlassen, wenn der String komplett analysiert ist. Das ist der Fall, wenn <j> auf das Trennzeichen nach dem letzten Kommando zeigt, das zugleich das letzte Zeichen des gesamten Strings ist. Bis dahin wird die Analyse fortgesetzt, da die Schleifenbedingung *While j <> Len(Menue$(Menue))* noch erfüllt ist.

Das Vortasten läuft so ab: <j> zeigt auf das Ende des zuletzt behandelten Kommandos, auf »#«. <i> ist eine Art »Hilfsvariable« und wird jeweils zu Beginn der Schleife mit j + 1 initialisiert, der Position des Anfangsbuchstabens des nächsten Kommandos. <MBefehle$> wird um diesen Buchstaben erweitert (*MBefehle$ = MBefehle$ + Mid$(Menue$(Menue), i, 1)*).

Dann wird ab dieser Position mit INSTR nach dem Zeichen »#« gesucht, also nach dem Kommandoende. Die gefundene Position wird dem

»Kommandoende-Zeiger« <j> zugewiesen (*j = Instr(i, Menue$(Menue), "#")*).

Wenn die Kommandolänge – die Differenz zwischen dem Kommandoende <j> und dem Anfang <i> – größer ist als der bisherige Wert von <WBreite>, wird <WBreite> diese neue »größte Länge« zugewiesen. Nach dem Verlassen der Schleife enthält <WBreite> die Länge des längsten Kommandos und damit die »Innenbreite« des Menüs. Unter »Innenbreite« ist die Breite ohne die Leerzeichen und die Rahmenzeichen rechts beziehungsweise links der Kommandos zu verstehen.

Nach dem Verlassen der Schleife wird zur Kommandobreite <WBreite> zweimal der Abstand <WOffset> zwischen den Kommandos und dem Rahmen addiert. Nun enthält <WBreite> die echte Menübreite.

Etwas mysteriös erscheint Ihnen sicher die Anweisung *Call UpLowCase(MBefehle$)*. UPLOWCASE wandelt alle Zeichen des übergebenen Strings in Kleinbuchstaben um. Vor diesem Aufruf enthält <MBefehle$> die großgeschriebenen Anfangsbuchstaben der Kommandos, danach die zugehörigen Kleinbuchstaben. Für Menü Nummer 2 unseres Demo-Pull-down-Menüs gilt:

```
Menue$(2)="Markieren#Löschen#-#Kopieren#Verschieben#"
```

<MBefehle$> vor Aufruf von UPLOWCASE: "MLKV"
<MBefehle$> nach Aufruf von UPLOWCASE: "mlkv"

Der Vorteil der Umwandlung: Die Direktanwahl eines Kommandos erfolgt über den Anfangsbuchstaben, und zwar ohne SHIFT, also kleingeschrieben. Da <MBefehle$> nun die kleingeschriebenen Anfangsbuchstaben enthält, kann ohne jede Umwandlung direkt mit INSTR geprüft werden, ob die gedrückte Taste in <MBefehle$> enthalten ist.

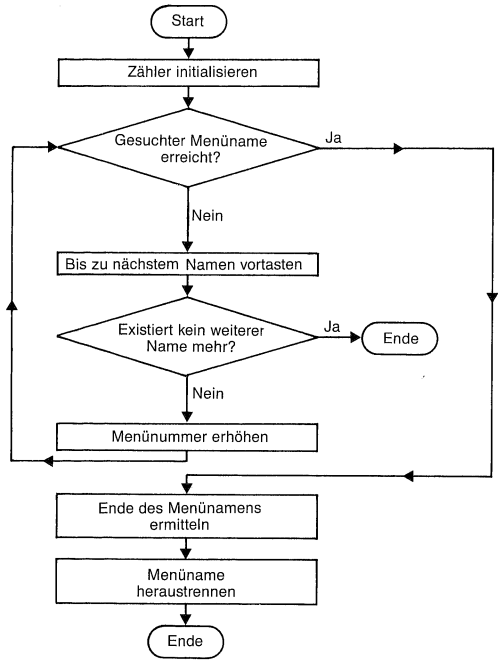


Bild 8 *Menüname ermitteln (GETNAME)*

GETNAME wird die Nummer eines Menüs übergeben. Die Prozedur ermittelt den zugehörigen Menünamen und seine Position innerhalb des Leistenstrings (genauer: die Position des Anfangsbuchstabens). Eine Erläuterung der wenigen Parameter ist überflüssig. Der »Prozedurkopf« beschreibt sie recht ausführlich.

```
'       GetName
'       -------
'       Funktion: Übergibt einzelnen Namen aus Menüheader
'       Hin     : StringNr : Nummer des gesuchten Strings
'       Zurück  : S$       : String
'                 SCol     : Startspalte des Strings
```

GETNAME arbeitet ähnlich wie INITMENUE. Die Prozedur tastet sich in einer Schleife im Leistenstring vorwärts. Dieses Vortasten ist beendet, wenn der gesuchte Menüname gefunden ist. Dem aufrufenden Programm wird der Name selbst und seine Position übergeben.

```
'       GetName
'       -------
'       Funktion: Übergibt einzelnen Namen aus Menüheader
'       Hin     : StringNr : Nummer des gesuchten Strings
```

```
'        Zurück   : S$        : String
'                   SCol      : Startspalte des Strings

Sub GetName(StringNr, S$, SCol) Static
  Nr   = Ø
  SCol = 1
  While Nr <> StringNr
    SCol = Instr(SCol, Header$, " ")
    Call SearchOther(Header$, " ", SCol)
    If SCol = Ø Then Exit Sub
    Nr = Nr + 1
  Wend
  ECol  = Instr(SCol, Header$, " ")
  S$    = Mid$(Header$, SCol, ECol - SCol)
End Sub
```

<Nr> ist in dieser Prozedur eine Zählvariable, die die Nummer des aktuellen Menünamens enthält (zu Beginn 0). <SCol> ist ein Zeiger auf die aktuelle Position im Leistenstring (zu Beginn 1 = erstes Zeichen). Die folgende Schleife wird erst verlassen, wenn <Nr> gleich <StringNr> ist, gleich der Nummer des gesuchten Menünamens.

In der Schleife tastet sich die Prozedur auf ähnliche Weise wie INITMENUE zum jeweils nächsten Namen vor. Zuerst muß der aktuelle Menüname »überlesen« werden.

Ausgehend von der aktuellen Position <SCol> wird mit der INSTR-Funktion das nächste Leerzeichen gesucht und diese Position <SCol> als neuer Wert zugewiesen (*SCol = Instr(SCol, Header$, " ")*). <SCol> weist nun unmittelbar hinter den aktuellen Menünamen, auf das folgende Leerzeichen.

Da einem Menünamen beliebig viele Leerzeichen folgen können, wird mit SEARCHOTHER nach dem nächsten Nicht-Leerzeichen gesucht (*Call SearchOther(Header$, " ", SCol)*), um alle Leerzeichen vor dem nächsten Namen zu überlesen.

Daß SEARCHOTHER nur noch Leerzeichen findet, in <SCol> den Wert 0 übergibt und die Prozedur vorzeitig verlassen wird (*If SCol = 0 Then Exit Sub*), tritt nur in einem Spezialfall auf: wenn der untersuchte String Nummer <StringNr> gar kein Menüstring ist und keine Kommandos enthält. Warum dieser Spezialfall getestet wird, sehen wir später. Behalten Sie diese Eigenschaft von GETNAME aber bis dahin in Erinnerung!

<SCol> zeigt nun auf das erste Zeichen des nächsten Menünamens, außer in einem Spezialfall: Möglicherweise ist das Stringende bereits erreicht, und ab der aktuellen Position gibt es daher kein Leerzeichen mehr. Um einen Laufzeitfehler zu vermeiden, wird die Prozedur in diesem Fall verlassen – ein weiteres Durchsuchen des Strings ist nicht möglich (*If SCol = 0 Then Exit Sub*).

Im Normalfall wird <Nr> um 1 erhöht. Das »Spielchen« wiederholt sich, bis <Nr> mit der Nummer des gesuchten Namens identisch ist.

Ab der aktuellen Position <SCol> (=Anfangsbuchstabe des gesuchten Namens) wird mit INSTR nach dem nächsten Leerzeichen gesucht. Der Menüname wird »überlesen« (*ECol = Instr(SCol, Header$, " ")*). <ECol> enthält nun die »Endspalte« des Menünamens, genauer: die Position des ersten folgenden Leerzeichens, also die Position des letzten Zeichens plus 1.

Dem aufrufenden Programm wird in <S$> der Menüname und in <SCol> die Startspalte übergeben. <SCol> enthält bereits die Startspalte. Zum Abschluß wird <S$> noch der Menüname zugewiesen, der mittlere Teil von <Header$> ab der Startspalte <SCol> in der Länge des Menünamens (*S$ = Mid$(Header$, SCol, ECol − SCol)*).

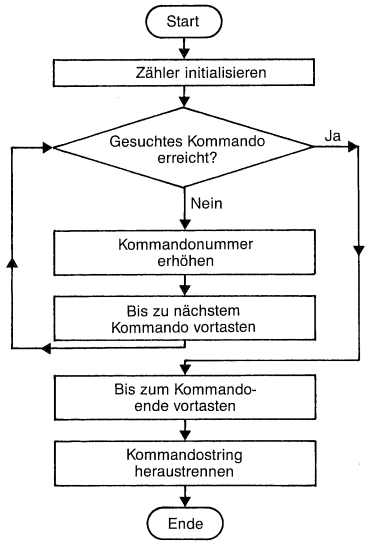


Bild 9 *Menükommando ermitteln (GETSTRING)*

GETSTRING ermittelt in einem Menüstring ein bestimmtes Kommando. Das aufrufende Programm übergibt in <Menue> die Nummer des Menüs und in <StringNr> die Nummer des gesuchten Kommandos.

```
'       GetString
'       ---------
'       Funktion: Übergibt einzelnen String des angegebenen Menüs
'       Hin     : Menue    : Nr. des aktiven Menüs
'                 StringNr : Nr. des gesuchten Menüstrings
'       Zurück  : S$       : String
```

GETSTRING detailliert zu erläutern, dürfte ein wenig langweilig für Sie sein. Erstens bietet diese Routine kaum Details (dafür ist sie zu einfach), zweitens ist sie mal wieder genauso aufgebaut wie INITMENUE und nun auch GETNAME.

```
Sub GetString(Menue, StringNr, S$) Static
  Nr = 1
  i  = Ø
  While Nr <> StringNr
    Nr = Nr + 1
    i  = Instr(i + 1, Menue$(Menue), "#")
  Wend
  j  = Instr(i + 1, Menue$(Menue), "#")
  S$ = Mid$(Menue$(Menue), i + 1, (j - 1) - i)
End Sub
```

<Nr> ist wieder die aktuelle Kommandonummer, <i> der Zeiger auf die aktuelle Position im Menüstring <Menue$(Menue)>. Die Schleife hangelt sich entlang der Trennzeichen »#« im Menüstring voran (*i = Instr(i + 1, Menue$(Menue), "#")*). Jedes Trennzeichen kennzeichnet das Ende eines Kommandos und den Beginn eines neuen Kommandos. <Nr> wird jeweils erhöht, bis der gewünschte Kommandostring Nummer <StringNr> erreicht ist (*Nr = Nr + 1*).

Ähnlich wie zuvor bei GETNAME wird ein letztes Mal INSTR angewendet, um das nächste Trennzeichen zu finden, also das Ende des gesuchten Kommandos. Nachdem Anfang und Ende feststehen, wird der betreffende Stringabschnitt »S$« zugewiesen.

Vielleicht fallen Ihnen einige kleine Unterschiede gegenüber GETNAME auf. Die beiden letzten Befehle verwenden statt der Startspalte <i> den Wert i + 1. Der Grund: <i> ist nicht die echte Startspalte eines Menünamens, sondern die Position des Trennzeichens unmittelbar vor dem ersten Buchstaben. Die echte Startspalte ist daher i + 1.

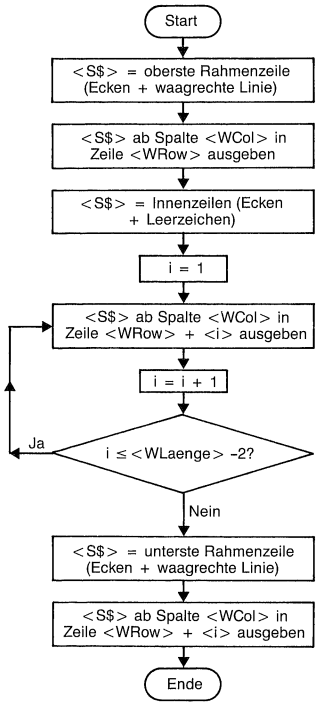


Bild 10 *Menü ausgeben (Teil 1, RAHMEN)*

Wir kümmern uns nun um die komplette Ausgabe eines Menüs. Komplett heißt, daß die Kommandos in einen Rahmen aus den Grafikzeichen des PCs »verpackt« werden.

Zu diesem Programmteil gibt es eine nette Vorgeschichte, die recht lehrreich ist. Etwa eine Stunde vor dem Schreiben dieser Zeilen genügte zur Menüausgabe noch eine Prozedur namens PRINTMENUE. Diese Prozedur wurde »soeben« komplett umgebaut und in zwei Prozeduren aufgeteilt – obwohl sie einwandfrei funktionierte!

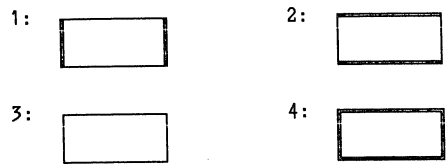
Die erste Version gab in einem Zug den Rahmen mit den darin enthaltenen Kommandos aus. Leider mußte ich bei der Arbeit an anderen Programmen feststellen, daß ich eine Routine zum Rahmenzeichnen ziemlich oft benötige.

PRINTMENUE war jedoch nicht allgemein genug, Rahmen beliebiger Art auszugeben. Die Funktion dieser Routine war auf die Ausgabe eines Menüs begrenzt. Die einzige vernünftige Lösung bestand in einem »Komplettumbau«, dem Auftrennen von PRINTMENUE in zwei unabhängige Routi-

nen. Eine, die einen Rahmen beliebiger Art zeichnet, und eine zweite, speziellere, die in diesem Rahmen Menükommandos ausgibt.

Ich erzähle diese Geschichte, damit Sie aus meinen Fehlern lernen können. Überlegen Sie sich bei jeder Prozedur zweimal, ob sie nicht zum Großteil auch in anderen Programmen benötigt wird. Wenn ja, achten Sie von vornherein auf Flexibilität!

Behandeln wir zuerst die Prozedur RAHMEN. RAHMEN ist tatsächlich sehr flexiblel. So flexibel, daß die Prozedur leider umfangreicher wurde als geplant. RAHMEN kann vier »Rahmentypen« verarbeiten und in verschiedenen Darstellungsarten ausgeben (normal, invers, intensiv etc.). Jeder Rahmentyp besitzt eine Nummer zwischen 1 und 4.

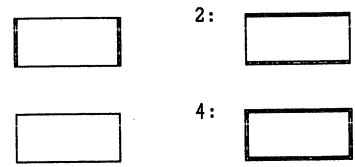


Beim Aufruf wird RAHMEN außer den Menüparametern (die nun als Rahmen-Parameter interpretiert werden) die gewünschte Darstellungsart und der Rahmentyp übergeben:

```
Sub Rahmen(WCol, WRow, WBreite, WLaenge, Attribut, Typ) Static
```

<WCol>, <WRow>, <WBreite> und <WLaenge> sind die von INITMENUE ermittelten Menüparameter. <Attribut> ist die gewünschte Darstellungsart, zum Beispiel einer der vordefinierten Typen <Normal>, <Invers> oder <Intensiv> (siehe COMDEF.BAS und INIT.BAS). <Typ> ist die »Rahmennummer« (1 bis 4, siehe Schema).

```
'       Rahmen zeichnen
'       ---------------

'       Funktion: Legt Rahmen um übergebenes Rechteck
'       Hin     : WCol/WRow/WBreite/WLaenge : Rechteck-Parameter
'                 Attribut : Darstellungsart
'                 Typ : 1-4, je nach Rahmentyp

'                       1:                2:
'
'

'                       3:                4:
'
'
```



```
Sub Rahmen(WCol, WRow, WBreite, WLaenge, Attribut, Typ) Static
```

```
'        * Rahmenzeichen *
  Dim Rahmen$(4)
  Rahmen$(1) = "┌┐└┘─│"
  Rahmen$(2) = "┌┐└┘═│"
  Rahmen$(3) = "┌┐└┘─│"
  Rahmen$(4) = "┌┐└┘═│"
```

Am Anfang der Prozedur werden die Strings <Rahmen$(1)> bis <Rahmen$(4)> definiert. Jeder String enthält die für einen Rahmentyp benötigten sechs Grafikzeichen, und zwar immer in der gleichen Reihenfolge:

1. Zeichen: Linke obere Ecke
2. Zeichen: Rechte obere Ecke
3. Zeichen: Linke untere Ecke
4. Zeichen: Rechte untere Ecke
5. Zeichen: Waagrechte Linie
6. Zeichen: Senkrechte Linie

Der Rahmen wird in drei Schritten gemalt. Zuerst gibt RAHMEN die oberste Zeile aus (┌────────┐).

```
'        * Oberste Zeile *
  S$ = Left$(Rahmen$(Typ), 1) +_
    String$(WBreite - 2, Mid$(Rahmen$(Typ), 5, 1)) +_
    Mid$(Rahmen$(Typ), 2, 1)
  Call PrintF(WCol, WRow, S$, Attribut)
```

<S$> wird das Zeichen ganz links (=linke obere Ecke) aus jenem String zugewiesen, der die Zeichen des betreffenden Rahmentyps enthält (*Left$(Rahmen$(Typ), 1)*).

An dieses Eckzeichen wird die waagrechte Rahmenlinie in der Gesamt-Rahmenbreite minus zwei angehängt (abzüglich der beiden Eckzeichen). Die waagrechte Linie ist das fünfte Zeichen im Rahmenstring Nummer <Typ> (*String$(WBreite – 2, Mid$(Rahmen$(Typ), 5, 1)*)

<S$> wird um die rechte obere Ecke ergänzt (zweites Zeichen der Rahmenstrings) und enthält die vollständige oberste Rahmenzeile. PRINTF gibt diesen String ab der Position der linken oberen Ecke des Rahmens (<WCol>/<WRow>) in der übergebenen Darstellungsart <Attribut> aus.

Im zweiten Schritt gibt RAHMEN alle folgenden Zeilen bis zur letzten aus (│ │).

```
'        * Mittlere Zeilen *
  S$ = Right$(Rahmen$(Typ),1) + Space$(WBreite-2) + Right$(Rahmen$(Typ), 1)
  For i=1 to WLaenge - 2
    Call PrintF(WCol, WRow + i, S$, Attribut)
  Next i
```

Wieder wird in <S$> eine komplette Rahmenzeile zusammengesetzt. Zuerst eine waagrechte Linie (letztes Zeichen der Rahmenstrings), dann ein String aus Leerzeichen, dessen Länge der Rahmenbreite abzüglich der beiden Eckzeichen entspricht. Das letzte Zeichen in <S$> ist wieder die waagrechte Linie des Rahmentyps Nummer <Typ> (*Right$(Rahmen$(Typ), 1)*).

<S$> wird nun mehrmals ausgegeben. Wie oft, hängt von der Länge <WLaenge> des Rechtecks ab (»WLaenge – 2« – mal). Die Ausgabe erfolgt immer ab der Startspalte <WCol>, jedoch in jedem Schleifendurchgang eine Zeile tiefer als zuvor (*WRow + i*).

Die letzte Zeile wird genauso ausgegeben wie die erste. Die Unterschiede: <S$> enthält außer der waagrechten Linie diesmal die Zeichen für die untere (!) rechte/linke Rahmenecke.

Und die Ausgabe erfolgt nicht in der Zeile <WRow>, sondern in <i>. <i> war die Schleifenvariable bei der Ausgabe der »mittleren« Zeilen. Der Wert von <i> ist nach Verlassen der Schleife um genau eins höher als bei der Ausgabe der letzten »mittleren« Zeile und somit genau richtig für die letzte Zeile.

```
'        * Unterste Zeile *
  S$ = Mid$(Rahmen$(Typ), 3, 1) +_
    String$(WBreite - 2, Mid$(Rahmen$(Typ), 5, 1)) +_
    Mid$(Rahmen$(Typ), 4, 1)
  Call PrintF(WCol, WRow + i, S$, Attribut)
End Sub
```

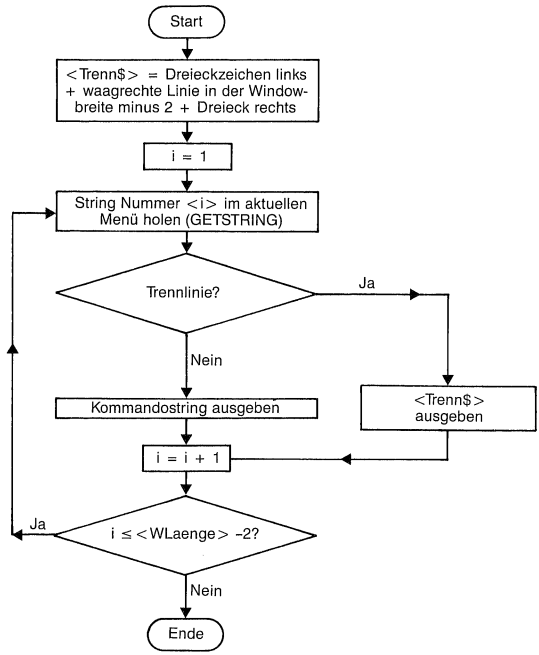


Bild 11 *Menü ausgeben (Teil 2, PRINTMENUE)*

Ich gebe zu: Der letzte »Brocken« war etwas schwer verdaulich, da der Umgang mit den vier Rahmenstrings ein klein wenig trickreich ist.

Dafür können Sie RAHMEN nicht nur zur Ausgabe von Pull-down-Menüs verwenden, sondern für Rahmen beliebiger Art!

Ein Rahmen macht noch kein Menü. Wir benötigen eine zweite Routine, PRINTMENUE. PRINTMENUE übergeben wir die Nummer <Menue> des auszugebenden Menüs und die von INITMENUE ermittelten Menüparameter (<WCol>, die »Startspalte«; <WBreite>, die Menübreite und <WLaenge>, die Menülänge).

```
'       PrintMenue
'       ----------
'       Funktion: Malt Menü
'       Hin     : Menue  : Nr. des aktiven Menüs
'                 WCol/WBreite/WLaenge : Menüparameter

Sub PrintMenue(Menue, WCol, WBreite, WLaenge) Static

'       * Rahmen zeichnen *
Call Rahmen(WCol, 2, WBreite, WLaenge, Intensiv, 1)
```

PRINTMENUE ruft RAHMEN mit diesen Parametern auf, die ja gleichzeitig den Menürahmen definieren. Bei dieser Definition fehlt jedoch noch die Zeile der linken oberen Ecke. PRINTMENUE übergibt für diesen Parameter den Wert 2. Die Menüs sollen ja direkt unterhalb der Leiste erscheinen (Zeile 2).

Als Rahmentyp wird 1 angegeben, ein Rahmen mit doppelten senkrechten und einfachen waagrechten Linien.

Nach der Ausgabe des Rahmens »füllt« ihn PRINTMENUE mit den Kommandos (beziehungsweise den Trennlinien).

```
'        * Window-Innenzeilen *
  Trenn$ = "╟" + String$(WBreite - 2, "-") + "╢"
  For i=1 to WLaenge - 2
    Call GetString(Menue, i, S$)
    If S$ <> "-" Then
      Call PrintF(WCol + WOffset, i + 2, S$, Intensiv)
    Else
      Call PrintF(WCol, i + 2, Trenn$, Intensiv)
    End If
 Next i
End Sub
```

<Trenn$> wird für die Ausgabe von Trennlinien benötigt. Eine Trennlinie besteht aus den Eckzeichen des Rahmentyps 1 (»╟« und »╢«) und der zugehörigen waagrechten Linie (»–«) in der Gesamtbreite des Rahmens abzüglich 2 ("╟" + *String$(WBreite – 2, "–")* + "╢").

Die Schleifenvariable <i> gibt die aktuelle Ausgabezeile an (i + 2). Nach dem Aufruf von GETSTRING enthält <S$> das Kommando Nummer <i>. Enthält <S$> ein normales Kommando und keinen Bindestrich (»-«), wird dieses Kommando in der Zeile *i + 2* ab Spalte *WCol + WOffset* (Startspalte des Rahmens plus Abstand zu den Kommandos) in intensiver Darstellung ausgegeben.

Ein Bindestrich als »Kommando« zeigt eine Trennlinie an. <Trenn$> – die Trennlinie – wird in Zeile *i + 2* ab der Startspalte des Rahmens <WCol> ebenfalls intensiv ausgegeben.

Auf die gleiche Weise behandelt PRINTMENUE auch die folgenden Zeilen. Jede »Rahmen-Innenzeile« ist nun mit einem Kommando oder einer Trennlinie gefüllt und das Menü vollständig.

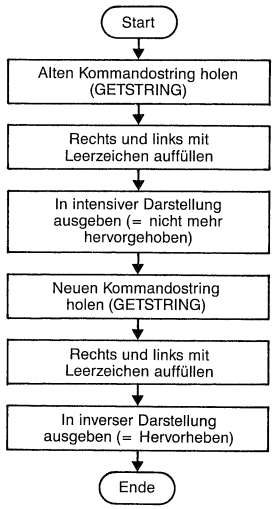


Bild 12 *Neues Kommando hervorheben (BEFEHLNEU)*

NAMENEU wird <WCol> (Startspalte der Kommandos), <WBreite> (maximale Kommandolänge), <CrsRowAlt> und <CrsRow> übergeben. <CrsRowAlt> ist – innerhalb des Menüs! – die Zeile des momentan invertierten Kommandos. NAMENEU soll dieses Kommando wieder »normalisieren«. <CrsRow> ist jene Menüzeile, in der sich das selektierte Kommando befindet. Dieses Kommando invertiert NAMENEU.

```
'       BefehlNeu
'       ---------
'       Funktion: – selektierten Befehl normalisieren
'                 – neu selektierten Befehl invertieren
'       Hin     : Menue     : Nr. des aktiven Menüs
'                 CrsRowAlt : Bisherige Cursorzeile im aktiven Menü
'                 CrsRow    : Voraussichtl.neue Cursorzeile im aktiven Menü
'                 WCol      : Startspalte des Menüs
'                 WBreite   : Breite des Menüs


Sub BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite) Static

'       * Alte Cursorzeile normalisieren *
  Call GetString(Menue, CrsRowAlt, S$)
  S$ = Space$(WOffset – 1) + S$ + Space$(WOffset – 1)
  If Len(S$) < WBreite – 2 Then S$ = S$ + Space$((WBreite – 2) – Len(S$))
  Call PrintF(WCol + 1, 2 + CrsRowAlt, S$, Intensiv)
```

Zuerst wird das »alte« Kommando normalisiert. GETSTRING übergibt in <S$> den String Nummer <CrsRowAlt> im Menü Nummer <Menue>. <S$> wird rechts und links mit Leerzeichen aufgefüllt, da die gesamte momentan inverse Zeile zwischen den beiden Rahmenbegrenzungen in nor-

maler Darstellung auszugeben ist. Links werden zwei Leerzeichen angefügt, die Anzahl der Leerzeichen rechts von <S$> hängt von der Kommandolänge ab.

<S$> wird nun in normaler Darstellung ausgegeben. <CrsRowAlt> ist die Zeilennummer innerhalb des Menüs. Da sich das erste Kommando bereits in der dritten Bildschirmzeile befindet, wird der String in der Zeile Nummer *2 + CrsRowAlt* ausgegeben.

Die Ausgabe beginnt ab der Startspalte plus 1, ein Zeichen rechts von der linken Rahmenbegrenzung (*WCol + 1*).

```
'        * Neue Cursorzeile invertieren *
  Call GetString(Menue, CrsRow, S$)
  S$ = Space$(WOffset - 1) + S$ + Space$(WOffset - 1)
  If Len(S$) < WBreite - 2 Then S$ = S$ + Space$((WBreite - 2) - Len(S$))
  Call PrintF(WCol + 1, 2 + CrsRow, S$, Invers)
End Sub
```

Die »Invertierung« der neu selektierten Zeile Nummer <CrsRow> läuft genauso ab. Diesmal wird <S$> jedoch zur Hervorhebung invers ausgegeben.

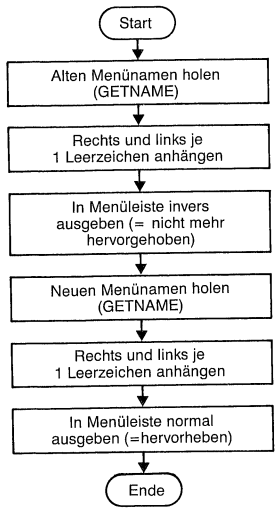


Bild 13 *Neuen Menünamen hervorheben (NAMENEU)*

NAMENEU hebt einen Menünamen in der Leiste hervor. Hervorheben heißt übrigens nicht, daß der Name invers dargestellt wird. Da die gesamte Leiste invers ist, wird ein einzelner Name durch Normaldarstellung hervorgehoben.

NAMENEU invertiert also den momentan normalisierten Namen und normalisiert den neuen Menünamen. Das aufrufende Programm übergibt NAMENEU <MenueAlt> (Nummer des bisher aktiven Menüs) und <Menue> (die Nummer des neuen aktiven Menüs).

```
'       NameNeu
'       -------
'       Funktion: - invertiert aktuellen Menünamen
'                 - normalisiert neuen Menünamen
'       Hin     : MenueAlt : Nr. des bisher aktiven Menüs
'                 Menue    : Nr. des neuen aktiven Menüs


Sub NameNeu(MenueAlt, Menue) Static

'       * Aktuellen Menünamen invertieren *
  Call GetName(MenueAlt, S$, SCol)
  S$ = " " + S$ + " "
  Call PrintF(SCol - 1, 1, S$, Invers)
```

Um den aktuellen Menünamen zu invertieren, muß die Prozedur wissen, wo sich dieser Name innerhalb der Menüleiste befindet. GETNAME wird aufgerufen und die alte Menünummer <MenueAlt> übergeben.

GETNAME sucht den betreffenden Namen in <Header$> und übergibt in <SCol> die Spaltenposition und in <S$> den Namen selbst.

In der Menüleiste wird nicht nur der eigentliche Menüname, sondern – optisch ansprechender – davor und danach noch je ein Leerzeichen hervorgehoben. <S$> wird um diese Leerzeichen ergänzt (*S$ = " " + S$ + " "*), mit dem Attribut <Invers> ausgegeben und der alte Menüname nun ebenso invers dargestellt wie der Rest der Leiste. Die Ausgabe beginnt nicht *SCol – 1*, da vor dem eigentlichen Namen ein Leerzeichen auszugeben ist.

```
'       * Neuen Menünamen invertieren *
  Call GetName(Menue, S$, SCol)
  S$ = " " + S$ + " "
  Call PrintF(SCol - 1, 1, S$, Normal)
End Sub
```

Das Hervorheben des neuen Menünamens verläuft analog. Die Unterschiede: GETNAME wird diesmal die Nummer des neuen aktiven Menüs <Menue> übergeben und <S$> nicht invers, sondern normal ausgegeben. <S$> hebt sich nun deutlich vom inversen Rest der Menüleiste ab.

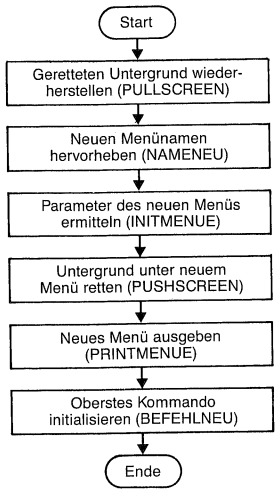


Bild 14 *Aktivierung eines Menüs (MENUENEU)*

Wir kommen nun zu den schwersten »Brocken« des gesamten Programms. MENUENEU steuert alle Aktivitäten, die bei der Aktivierung eines neuen Menüs nötig sind:

- Die Wiederherstellung des Untergrunds unter dem momentan noch aktiven Menü.
- Das Hervorheben des Namens des neuen Menüs in der Leiste.
- Das Ermitteln der Parameter des neuen Menüs und das Retten des aktuellen Untergrunds vor dem »Aufklappen«.
- Die Ausgabe des neuen Menüs und das Hervorheben des obersten darin enthaltenen Kommandos.

Alle diese Aktivitäten erledigt MENUENEU ausschließlich durch das Aufrufen der bereits bekannten Prozeduren in einer sinnvollen Reihenfolge!

»Oberhalb« von MENUENEU befindet sich nur die Kommandoschleife, die die Eingaben des Benutzers überwacht und die jeweils passende Prozedur aufruft. MENUENEU ruft dieses Steuerungsprogramm auf, wenn der Benutzer CURSOR RECHTS oder CURSOR LINKS drückt oder aber (SHIFT + Anfangsbuchstabe) ein Menü direkt anwählt. MENUENEU wartet mit einer stattlichen Anzahl an Parametern auf:

```
'       MenueNeu
'       --------
'       Funktion: Behandelt Anwahl eines neuen Menüs
'       Hin     : MenueAlt  : Nr. des bisher aktiven Menüs
'                 Menue     : Nr. des neuen aktiven Menüs
'                 CrsRow    : Nr. der bisher selektierten Menüzeile
'                 WCol/WBreite/WLaenge : Parameter des alten Menüs
'                 MBefehle$ : Anfangsbuchstaben (klein) der Menübefehle
'       Zurück  : CrsRow    : Nr. der neuen selektierten Menü-
                              zeile (immer 1)
'                 WCol/WBreite/WLaenge : Parameter des neuen Menüs
```

Alle Parameter kennen Sie aus den bisher erläuterten Prozeduren. Beschäftigen wir uns lieber mit dem Ablauf. Zuerst muß das gerade aktive Menü zugeklappt werden, das heißt, der alte Untergrund wird wiederhergestellt.

Für das Retten und Wiederherstellen eines Untergrundes sind die Prozeduren PUSHSCREEN und PULLSCREEN zuständig, die wir erst ganz am Schluß dieses Programms näher kennenlernen. Der Grund ist ganz einfach: Beide Prozeduren sind sehr komplex und besser zu verstehen, wenn Sie mit dem Gesamtprogramm vertraut sind. PUSHSCREEN wird so aufgerufen:

```
Call PushScreen(WCol, WRow, WBreite, WLaenge)
```

<WCol>, <WRow>, <WBreite> und <WLaenge> definieren den zu rettenden Bildschirmausschnitt, die Position der linken oberen Ecke, die Breite und die Länge des Rechtecks. Diese Parameter entsprechen den von INITMENUE ermittelten Menüparametern. <WRow>, die oberste Zeile, ist bei unseren Pull-down-Menüs immer 2.

PULLSCREEN kopiert gerade umgekehrt einen geretteten Ausschnitt wieder auf den Bildschirm zurück und stellt den Original-Bildschirminhalt wieder her:

```
Call PullScreen(WCol, WRow, WBreite, WLaenge)
```

Der Aufruf ist identisch. Auch PULLSCREEN muß nur wissen, welcher Ausschnitt wiederherzustellen ist und benötigt sonst keine Angaben.

Kommen wir wieder zu MENUENEU. Der vom momentan aktiven Menü überschriebene Bildschirmausschnitt muß wiederhergestellt werden.

```
Sub MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge, MBefehle$)
Static

'       * Alten Untergrund holen *
  Call PullScreen(WCol, 2, WBreite, WLaenge)
```

Die Zeile, in der sich die obere linke Ecke befindet, ist immer Zeile Nummer 2, eine Zeile unterhalb der Menüleiste. Um die restlichen Parameter müssen

wir uns nicht weiter kümmern. Sie wurden bereits von MENUEINIT ermittelt.

Nach dem Herstellen des Original-Untergrundes ist das bisher aktive Menü »zugeklappt«. Nun hebt MENUENEU den Namen des neuen aktiven Menüs in der Leiste hervor.

```
'        * Hervorheben eines anderen Menünamens *
  Call NameNeu(MenueAlt, Menue)
```

NAMENEU stellt den bisher hervorgehobenen Menünamen Nummer <MenueAlt> ebenso invers dar, wie den Rest der Leiste und hebt dafür den Namen des neuen aktiven Menüs hervor, den Namen Nummer <Menue>.

Im dritten Schritt werden durch Aufruf von INITMENUE die Parameter des neuen Menüs ermittelt. Seinen aktuellen Untergrund rettet PUSHSCREEN und PRINTMENUE gibt das Menü aus.

```
'        * Neue Menüparameter holen und Untergrund retten *
  Call InitMenue(Menue, WCol, WBreite, WLaenge, MBefehle$)
  Call PushScreen(WCol, 2, WBreite, WLaenge)
  Call PrintMenue(Menue, WCol, WBreite, WLaenge)
```

Nach dem Aufruf von INITMENUE enthalten die Variablen <WCol>, <WBreite> und <WLaenge> nicht mehr die Parameter des alten, sondern die des neuen aktiven Menüs.

PUSHSCREEN rettet den Untergrund dieses neuen Menüs und PRINTMENUE gibt es auf dem Bildschirm aus – das Menü ist »aufgeklappt«.

Eine letzte Kleinigkeit fehlt noch. Es ärgert mich sehr, daß beim Aufklappen eines QuickBASIC-Menüs noch kein Kommando selektiert ist. Benutzerfreundlicher wäre es, wenn nach dem Aufklappen bereits das erste Kommando selektiert ist. Wenn Sie dieses Kommando anwählen wollen, drücken Sie einfach RETURN. Und zur Anwahl eines anderen Kommandos müssen Sie einmal weniger die Taste CURSOR UNTEN drücken.

Auch wenn die Programmierer von QuickBASIC diese Kleinigkeit vergaßen, unser Programm enthält sie selbstverständlich. Daher selektiert die Prozedur nun den ersten Befehl des aktiven Menüs.

```
'        * Befehl 1 selektieren *
  CrsRowAlt = 1: CrsRow = 1
  Call BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite)
End Sub
```

BEFEHLNEU normalisiert den Befehl Nummer <CrsRowAlt> und invertiert den Befehl Nummer <CrsRow>. Welcher Befehl ist der »alte«, der nun normalisiert werden soll? Es gibt keine Antwort darauf und sie ist auch

überflüssig. <CrsRowAlt> kann jeder beliebige Wert zwischen 1 und <WLaenge> zugewiesen werden (in diesem Program 1).

BEFEHLNEU normalisiert nun zwar den obersten Befehl des neuen Menüs. Da dieser Befehl bereits normal dargestellt ist, ergibt sich keine sichtbare Änderung. Wichtig ist nur die Initialisierung von <CrsRow> mit 1, damit BEFEHLNEU den ersten Befehl invertiert.

Das Steuerungsprogramm (PULLDOWN)

Wir haben uns inzwischen auf recht vielen Seiten durch die verschiedenen Ebenen von PULLDOWN gewühlt, gewissermaßen »von unten nach oben«. Zuerst lernten Sie die Prozeduren der untersten Ebene kennen (INITMENUE, GETNAME, GETSTRING, RAHMEN und PRINTMENUE), dann die etwas gewichtigeren BEFEHLNEU und NAMENEU und zuletzt die Prozedur MENUENEU, die im Grunde nur noch aus Aufrufen der »tieferen« Prozeduren besteht.

Inzwischen sind wir »ganz oben«. Es fehlt nur die eigentliche Hauptprozedur, die die Kommandos des Benutzers überwacht und den gesamten Ablauf steuert.

Der Aufruf von PULLDOWN lautet:

```
Sub PullDown(Menue, CrsRow) Static
```

<Menue> und <CrsRow> werden uns die Nummer des aktiven Menüs und des darin gewählten Kommandos mitteilen. Vor dem Aufruf muß die Menüleiste <Header$> und das Menüarray <Menue$(..)> definiert sein.

Bevor irgendein Menü verwaltet wird, gibt PULLDOWN die Menüzeile aus.

```
'       PullDown
'       --------
'       Funktion: Verwaltet Pull-down-Menüs
'       Zurück  : Menue    : Nr. des aktiven Menüs
'                 CrsRow   : Nr. des selektierten Befehls

Sub PullDown(Menue, CrsRow) Static

'       *** Initialisierungsteil ***

'       * Init Header *
  Call PrintF(1, 1, Header$, Invers)
```

Die Ausgabe erfolgt mit dem Attribut <Invers>. Die gesamte erste Zeile wird invers dargestellt.

Nun ermittelt PULLDOWN, wie viele Einzelmenüs das komplette Pulldown-Menü enthält und welche Anfangsbuchstaben die Menüs besitzen (für die Direktanwahl!).

```
'       * Menüanzahl und Menünamen ermitteln *
 MName$ = ""
 i = Ø
 SCol = 1
 While SCol <> Ø
   i = i + 1
   Call GetName(i, S$, SCol)
   MName$ = MName$ + Left$(S$, 1)
 Wend
 MZahl  = i - 1
 MName$ = Left$(MName$, Len(MName$) - 1)
```

<MName$> wird die Anfangsbuchstaben enthalten und <i> gibt die Nummer des aktuellen Menüs an. Die Methode zur Ermittlung der Menüanzahl ist einfach: Immer wieder wird GETNAME aufgerufen, um den Namen des Menüs Nummer <i> zu ermitteln (*Call GetName(i, S$, SCol)*). <MName$> wird jedesmal um den Anfangsbuchstaben des übergebenen Menünamens <S$> erweitert (*MName$ = MName$ + Left$(S$, 1)*).

Vor jedem Aufruf wird die Menünummer <i> um eins erhöht. Wenn das Menü Nummer <i> nicht existiert (der String <Menue$(i)> keine Kommandos enthält), übergibt GETNAME als Startspalte <SCol> den Wert 0. Dabei handelt es sich um jene Eigenschaft von GETNAME, die ich nicht näher erläutern wollte, die Sie jedoch in Erinnerung behalten sollten. Nun wissen Sie, warum!

Wenn <SCol> gleich 0 ist, sind wir also »über das Ziel hinausgeschossen«. <i> ist dann bereits größer als die Menüanzahl. Die Schleife wird verlassen, <MZahl> die korrekte Anzahl i – 1 zugewiesen und aus <MName$> das letzte Zeichen entfernt (der »Anfangsbuchstabe« des nicht existierenden Menüs).

Halten wir fest: <MName$> enthält nun die Anfangsbuchstaben aller Menüs und <MZahl> die Menüanzahl.

Im Gegensatz zu QuickBASIC klappt PULLDOWN nach dem Aufruf sofort das erste Menü auf. Dieses Menü wird nun initialisiert.

```
'       * Init Menue 1 *
 Menue = 1
 Call InitMenue(Menue, WCol, WBreite, WLaenge, MBefehle$)
 Call PushScreen(WCol, 2, WBreite, WLaenge)
 Call MenueNeu(Menue, Menue, CrsRow, WCol, WBreite, WLaenge, MBefehle$)
```

<Menue> wird die Nummer 1 zugewiesen und INITMENUE ermittelt die zugehörigen Parameter. MENUENEU übernimmt alle weiteren Vorbereitungen. Diese Prozedur geht jedoch von einem bereits aktiven Menü aus, dessen Untergrund gerettet wurde.

Da das momentan nicht der Fall ist, wird der Versuch von MENUENEU, den Untergrund mit der Prozedur PULLSCREEN wiederherzustellen, zu einem Fehler führen!

Daher rufen wir PUSHSCREEN auf, um den Untergrund des noch gar nicht aufgeklappten Menüs Nummer 1 zu retten.

Beim Aufruf von MENUENEU wird nun sowohl als Nummer des alten und auch des neuen Menüs die gleiche Nummer 1 angegeben.

Das heißt, MENUENEU holt den soeben geretteten Untergrund wieder zurück, hebt den Menünamen hervor, rettet den Untergrund und so weiter – Menü Nummer 1 ist aktiviert!

Nach dieser Initialisierung beginnt die »Hauptschleife«. In dieser Schleife wird auf ein Kommando des Benutzers gewartet und je nach Kommando ein neuer Menübefehl selektiert oder ein anderes Menü aktiviert. Die Schleife wird nach der Anwahl eines Kommandos verlassen. Der prinzipielle Aufbau:

```
ExitFlag = False
While ExitFlag = False
Call Taste(Key$, Escape)
  If Key$ = zReturn$ Then
    ...
  ElseIf Key$ = zRight$ Then
    ...
  ElseIf Key$ = zLeft$ Then
    ...
  ElseIf Instr(MName$, Key$) <> Ø Then
    ...
  ElseIf Key$ = zDown$ Then
    ...
  ElseIf Key$ = zUp$ Then
    ...
  ElseIf Instr(MBefehle$, Key$) <> Ø Then
    ...
  End If
Wend
```

<ExitFlag> gibt an, ob die Schleife beendet ist und wird mit <False> initialisiert. Der Aufruf von TASTE (in der Routinen-Sammlung) wartet auf eine Taste und übergibt sie in <Key$>.

Die folgenden IF-/ELSEIF-Anweisungen behandeln die zulässigen Tastenkommandos (Cursortasten etc.).

Bei dieser Behandlung wird <ExitFlag> der Wert <True> zugewiesen, wenn der Benutzer ein Menükommando anwählt. Das heißt, die Bedingung *While ExitFlag = False* ist nicht mehr erfüllt und die Schleife wird verlassen. Sehen wir uns nun der Reihe nach die Behandlung der verschiedenen Tasten an.

1. RETURN

```
'       *** Hauptschleife ***
  ExitFlag = False
  While ExitFlag = False
    Call Taste(Key$, Escape)
      If Key$ = zReturn$ Then
        ExitFlag = True
```

Enthält <Key$> als Taste <zReturn$>, drückte der Benutzer RETURN, um das momentan selektierte Kommando anzuwählen. <ExitFlag> wird der Wert <True> zugewiesen und die Schleife beim nächsten Durchgang verlassen.

2. CURSOR RECHTS

```
ElseIf Key$ = zRight$ Then
  MenueAlt = Menue
  Menue    = Menue + 1
  If Menue > MZahl Then Menue = 1
  Call MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge,_
  MBefehle$)
```

CURSOR RECHTS klappt das Menü auf, das sich rechts vom gerade aktiven befindet. In <MenueAlt> wird die aktuelle Menünummer <Menue> zwischengespeichert und die Menünummer um eins erhöht.

Möglicherweise ist <Menue> nun größer als <MZahl> (das heißt, das letzte Menü war bereits aktiv, als der Benutzer CURSOR RECHTS drückte). Wenn ja, erhält <Menue> den Wert 1, das erste Menü wird aktiviert.

Die eigentliche Aktivierung – das Zuklappen des alten Menüs <MenueAlt> und das Aufklappen des neuen aktiven Menüs Nummer <Menue> – übernimmt MENUENEU.

3. CURSOR LINKS

```
ElseIf Key$ = zLeft$ Then
  MenueAlt = Menue
  Menue    = Menue - 1
  If Menue = Ø Then Menue = MZahl
  Call MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge,_
  MBefehle$)
```

CURSOR LINKS wird analog behandelt. In <MenueAlt> wird die aktuelle Menünummer gespeichert, danach die Menünummer um eins vermindert. War bereits das erste Menü aktiv, enthält <Menue> nun den Wert 0 und wird auf <MZahl> gesetzt, die Nummer des letzten Menüs. Mit MENUENEU wird das alte zugeklappt und das neue Menü aktiviert.

4. Direkte Menüanwahl

```
ElseIf Instr(MName$, Key$) <> Ø Then
  MenueAlt = Menue
  Menue    = Instr(MName$, Key$)
  Call MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge,_
  MBefehle$)
```

<MName$> enthält der Reihe nach die – großgeschriebenen – Anfangsbuchstaben der Menüs 1 bis <MZahl>. Ist <Key$> mit einem dieser Buchstaben identisch, wählt der Benutzer ein Menü direkt an, ohne den Umweg über die Cursortasten. In <MenueAlt> wird wieder die bisherige Menünummer gespeichert und mit der INSTR-Funktion die Nummer des neuen Menüs ermittelt. Diese Nummer ist identisch mit der Position des betreffenden Anfangsbuchstabens in <MName$>. MENUENEU erledigt wie immer den Rest, das Zu- beziehungsweise Aufklappen der beiden Menüs.

5. CURSOR UNTEN

```
ElseIf Key$ = zDown$ Then
  CrsRowAlt = CrsRow
  CrsRow    = CrsRow + 1
  If CrsRow > WLaenge – 2 Then CrsRow = 1
  Call GetString(Menue, CrsRow, S$)
  If S$ = "-" Then CrsRow = CrsRow + 1
  Call BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite)
```

Die Behandlung von CURSOR UNTEN ist etwas umfangreicher. BEFEHLNEU behandelt die Selektion eines neuen Kommandos. Die Prozedur erwartet vor allem die Übergabe der Parameter <CrsRowAlt> (Nummer der momentan innerhalb des Menüs selektierten Zeile) und <CrsRow> (Nummer der nun zu selektierenden Zeile).

<CrsRowAlt> ist natürlich mit dem bisherigen Wert von <CrsRow> identisch. Die neu selektierte Zeile ist nach dem Kommando CURSOR UNTEN gleich dem alten Wert von <CrsRow> plus 1.

Zwei Spezialfälle sind zu beachten: Jedes Menü enthält *WLaenge – 2* Kommandos (Rahmenlänge minus 2). Ein größerer Wert von <CrsRow> bedeutet, daß ein nicht existierendes Kommando selektiert wird (das Kommando »unterhalb« des letzten Kommandos).

Das passiert, wenn das letzte Kommando selektiert ist und der Benutzer nun CURSOR UNTEN drückt. In diesem Fall erhält <CrsRow> den Wert 1, um das oberste Menükommando zu selektieren (*If CrsRow > WLaenge Then CrsRow = 1*).

Der zweite Spezialfall: Die nächstuntere Zeile enthält kein »echtes« Kommando, sondern eine Trennlinie zwischen zwei Kommandoblöcken. Eine sol-

che Trennlinie müssen wir übergehen! <CrsRow> wird ein weiteres Mal um 1 erhöht.

Die Prüfung auf eine Trennlinie erfolgt so: Nach dem Erhöhen von <CrsRow> wird GETSTRING aufgerufen. GETSTRING übergibt den String Nummer <CrsRow> im momentan aktiven Menü Nummer <Menue>. Enthält dieser String das Zeichen »-«, ist die Zeile Nummer <CrsRow> eine Trennlinie (*If S$ = "-" Then CrsRow = CrsRow + 1*). <CrsRow> wird ein zweites Mal um eins erhöht und »zeigt« nun auf das erste Kommando unterhalb der Trennlinie.

Alle möglichen Spezialfälle sind berücksichtigt und BEFEHLNEU wird aufgerufen. BEFEHLNEU normalisiert das bisher selektierte Kommando <CrsRowAlt> und invertiert das Kommando <CrsRow>.

6. CURSOR OBEN

```
ElseIf Key$ = zUp$ Then
  CrsRowAlt = CrsRow
  CrsRow    = CrsRow - 1
  If CrsRow = Ø Then CrsRow = WLaenge - 2
  Call GetString(Menue, CrsRow, S$)
  If S$ = "-" Then CrsRow = CrsRow - 1
  Call BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite)
```

CURSOR OBEN wird prinzipiell genauso behandelt. <CrsRow> wird diesmal nicht erhöht, sondern um eins vermindert. Die beiden Spezialfälle treten leicht variiert erneut auf.

Wenn <CrsRow> nach dem Vermindern um eins den Wert 0 besitzt, war das oberste Kommando selektiert, als der Benutzer die Taste CURSOR OBEN drückte. <CrsRow> soll nun auf das unterste Kommando zeigen und erhält den Wert *WLaenge - 2* (Rahmenlänge minus 2 = Nummer des letzten Kommandos).

Mit GETSTRING wird wieder geprüft, ob <CrsRow> auf eine Trennlinie zeigt und in diesem Fall <CrsRow> ein zweites Mal um eins vermindert.

7. Direktanwahl eines Kommandos

```
    ElseIf Instr(MBefehle$, Key$) <> Ø Then
      CrsRow   = Instr(MBefehle$, Key$)
      ExitFlag = True
    End If
  Wend
```

<MBefehle$> enthält die – kleingeschriebenen – Anfangsbuchstaben der Befehle des aktuellen Menüs. Wenn <Key$> in <MBefehle$> enthalten ist, wählt der Benutzer ein Kommando direkt an. Die Position von <Key$> in <MBefehle$> ist mit der Nummer des Kommandos identisch (denken Sie

daran: Die Menükommandos enthalten <MBefehle$>. Diese Kommandonummer wird in <CrsRow> gespeichert (*CrsRow = Instr(MBefehle$, Key$)*). <ExitFlag> wird der Wert <True> zugewiesen, um die Schleife zu verlassen.

»Echte« Befehlsnummer ermitteln

Alle möglichen Kommandos sind behandelt. Zuletzt wird die komplette Menüleiste invers ausgegeben, um zu verhindern, daß der zuletzt aktive Menüname weiterhin invers ist. Der Original-Untergrund des zuletzt aktiven Menüs wird wiederhergestellt und eine letzte Kleinigkeit erledigt:

<CrsRow> enthält die Nummer des – direkt oder mit RETURN – gewählten Kommandos. Aber nicht unbedingt die »echte« Nummer.

Erinnern Sie sich an die Trennlinien. Jede Trennlinie zählt ebenfalls als Kommando! <CrsRow> wird daher entsprechend korrigiert.

```
'       * Selektierten Befehl ermitteln *
  Call PrintF(1, 1, Header$, Invers)
  Call PullScreen(WCol, 2, WBreite, WLaenge)
  For i=1 to CrsRow
    Call GetString(Menue, i, S$)
    If S$ = "-" Then CrsRow = CrsRow – 1
  Next i
End Sub
```

In einer Schleife wird <CrsRow>-mal GETSTRING aufgerufen. Der übergebene Kommandostring Nummer <i> wird mit dem Symbol für eine Trennlinie verglichen. Ist er damit identisch, wird <CrsRow> um eins vermindert.

Befinden sich zum Beispiel zwei Trennlinien oberhalb des gewählten Kommandos, wird <CrsRow> in dieser Schleife zweimal um eins vermindert und enthält anschließend die »echte« Kommandonummer.

7.6 Statische und dynamische Arrays

Der folgende Abschnitt behandelt die zweifellos komplexesten Routinen von PULLDOWN. Komplex aufgrund der etwas verwickelten Datenstrukturen. Die besprochenen Routinen PUSHSCREEN und PULLSCREEN arbeiten mit einem »Pointer, der auf eine Tabelle zeigt, die wiederum Pointer auf ein Array enthält«. Klingt kompliziert, nicht wahr? Ist es leider auch.

Aber dafür lernen Sie in diesem Abschnitt auch einiges über QuickBASIC. Vor allem über die Vorzüge »dynamischer« gegenüber den bisher verwendeten »statischen« Arrays.

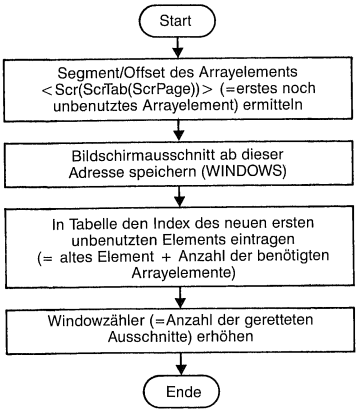


Bild 15 *Windowing-Routinen*

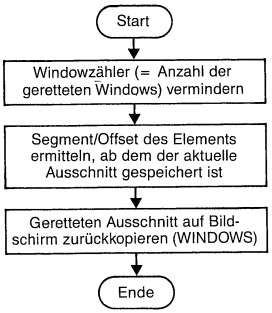


Bild 16 *Windowing-Routinen*

Das Retten/Wiederherstellen eines Menüuntergrundes übernehmen die Prozeduren PUSHSCREEN und PULLSCREEN. Beide Prozeduren rufen eine aus Geschwindigkeitsgründen in Assembler geschriebene Routine namens WINDOWS auf, die zwei Aufgaben erfüllt:

1. Sie kopiert den Inhalt eines beliebigen rechteckigen Bildschirmausschnitts – eines Windows – in ein Array.
2. Sie stellt den Inhalt des Windows wieder her, indem der Arrayinhalt in den Bildschirmausschnitt zurückkopiert wird.

Als Zwischenspeicher für den Ausschnitt wird das Integerarray <Scr(..)> verwendet (Integer wegen der Deklaration *DefInt a-z* in COMDEF.BAS). <Scr(..)> ist in COMDEF.BAS als globale Variable deklariert. <Scr(..)> und alle weiteren Window-Variablen werden in INIT.BAS initialisiert:

```
'        * Windowvariablen *
Dim Menue$(20)      'Anzahl der Pull-down-Menüs
MaxChar = 4000: Dim Scr(MaxChar)    'Pufferspeicher (dynamisches Array)
MaxTab  = 10:   Dim ScrTab(MaxTab) 'Window-Tabelle (dynamisches Array)
ScrPage   = 1       'Aktuelle Pufferseite
ScrTab(1) = 1       'Ersten Tabelleneintrag initialisieren
WOffset   = 3       'Abstand Window-Rahmen zum Window-Inhalt
```

Sie wundern sich vielleicht, daß die Arrays <Scr(..)> und <ScrTab(..)> scheinbar umständlich dimensioniert werden. Statt

```
MaxChar = 4000: Dim Scr(MaxChar)
```

sollte man auch direkt schreiben können

```
Dim Scr(4000)
```

Zwischen der Dimensionierung über eine Konstante und der Dimensionierung über eine Variable besteht jedoch ein gewaltiger Unterschied. Dimensionierungen mit Konstanten deklarieren ein Array als »statisch«. Statische Arrays speichert QuickBASIC in seinem »Datensegment«.

Das Datensegment ist ein 64 Kbyte großer Speicherbereich, der alle globalen Variablen und alle statischen Arrays aufnimmt. Wenn der Umfang dieser Variablen 64 Kbyte überschreitet, erhalten Sie den aus GW-BASIC bekannten »out of memory error«. Ihr Programm läuft nicht mangels Speicherplatz. Und das ist außerordentlich ärgerlich, wenn Ihr Rechner auf 512 Kbyte oder mehr ausgebaut ist und Sie genau wissen, daß momentan noch mehrere hundert Kbyte an freiem Speicherplatz vorhanden sind. Aber leider ist das Datensegment nun mal auf 64 Kbyte beschränkt. Dieser begrenzte Speicherplatz ist entsprechend kostbar und sollte nicht unnötig verschwendet werden.

Dimensionieren mit einer Variablen deklariert ein Array als »dynamisch«. Diese Arrays befinden sich nicht im eng begrenzten Datensegment. Dynamische numerische(!) Arrays können den gesamten verfügbaren Platz im Hauptspeicher verwenden (aber maximal 64 Kbyte pro Array). Daher sollten Sie größere numerische Arrays als dynamisch deklarieren, um keinen kostbaren Speicherplatz im Datensegment zu verschwenden.

Leider gilt das nur für dynamische numerische Arrays, nicht für dynamische Stringarrays. Nur dynamische numerische Arrays können den gesamten Rechnerspeicher nutzen.

Aber wie wir noch sehen werden, besitzen dynamische Arrays einen weiteren Vorteil – und der betrifft alle Typen dynamischer Arrays.

Der von einem dynamischen Array belegte Platz kann jederzeit wieder freigegeben werden (mit ERASE). Daher sollten Sie für »temporäre Arrays«, die Sie nur vorübergehend benötigen, immer dynamische Arrays verwenden.

Jede Integervariable besteht aus zwei Byte. Also bieten die 4000 Integervariablen von <Scr(..)> Platz für 8000 zu rettende Bytes. Der gesamte Bildschirm enthält 2000 Zeichen. 2000 Zeichen entsprechen 4000 Byte (zu jedem »Zeichenbyte« gehört ein »Attributbyte«). <Scr(..)> bietet also genug Platz zur gleichzeitigen Speicherung mehrerer Windows. Und da <Scr(..)> ein dynamisches Array ist, werden keine kostbaren 8000 Byte im Datensegment belegt (im Vergleich dazu macht die Deklaration der Tabelle <ScrTab(..)> als ebenfalls dynamisches Array kaum etwas aus).

Auf der Diskette zum Buch befindet sich sowohl das Source-Listing der Routine WINDOWS (unter dem Namen WINDOWS.ASM) als auch der Objektcode (WINDOWS.OBJ). Die Assembler-Routine wird im letzten Teil des Buches erläutert (»QuickBASIC und Assembler«). Uns interessiert an dieser Stelle nur die Anwendung der Routine.

WINDOWS erwartet die Übergabe der Parameter <Col>, <Row>, <Breite>, <Laenge>, <VarSeg>, <VarOff> und <Flag>.

```
Call Windows(Col, Row, Breite, Laenge, VarSeg, VarOff, Flag)
```

<Col>, <Row>, <Breite> und <Laenge> definieren das rechteckige Window, in unserem Fall ein Menü. <Flag> gibt an, ob der Ausschnitt in das Array <Scr(..)> gerettet (Flag = False) oder umgekehrt ein bereits in das Array kopierter Ausschnitt auf den Bildschirm zurückgeschrieben wird (Flag = True).

<VarSeg> und <VarOff> geben die Speicheradresse des Arrays an. Sollten Sie keine Erfahrung mit Assembler haben, fragen Sie sich bestimmt, wie Sie diese Speicheradresse ermitteln. Ganz einfach – mit der Prozedur PTR86.

PTR86 befindet sich auf der Original-QuickBASIC-Diskette unter dem Namen INT86.ASM (Sourcecode) beziehungsweise INT86.OBJ (Objektcode). Beide Objektdateien finden Sie auch auf der Diskette zum Buch. PTR86 übergibt die Speicheradresse einer angegebenen Variablen. Der Aufruf:

```
Call Ptr86(VarSeg, VarOff, VarPtr(Variable))
```

Ein Beispiel: Sie wollen den Inhalt des kompletten Bildschirms im Array <Scr(..)> »puffern«. WINDOWS müssen Variablen als Argumente übergeben werden. Daher definieren wir:

```
Col    = 1       'Spalte der linken oberen Ecke
Row    = 1       'Zeile der linken oberen Ecke
Breite = 80      'Breite des Windows
Laenge = 25      'Länge des Window
Flag   = False   'False => Window ins Array kopieren
```

Die Assembler-Routine soll die einzelnen Zeichen (und die zugehörigen Attribute) ab dem Arraybeginn speichern, ab der Variablen <Scr(1)>. Um WINDOWS die Adresse dieser Variablen übergeben zu können, wird sie zuvor mit PTR86 ermittelt.

```
Call Ptr86(VarSeg, VarOff, VarPtr(Scr(1)))
```

PTR86 übergibt die Adresse von <Scr(1)> in den Variablen <VarSeg> und <VarOff>. Nun können Sie WINDOWS aufrufen.

```
Call Windows(Col, Row, Breite, Laenge, VarSeg, VarOff, Flag)
```

Der gesamte Bildschirm enthält 2000 Zeichen. Diese 2000 Zeichen werden in den 2000 Variablen <Scr(1)> bis <Scr(2000)> gespeichert. In jeder Integervariablen (zwei Byte) wird jeweils ein Zeichen und das zugehörige Attribut (invers, blinkend etc.) gespeichert. Da 8000 Zeichen zur Verfügung stehen, können wir sogar mehrere überlappende Windows verwalten.

Da nach dem im Beispiel verwendeten Aufruf die Variablen <Scr(1)> bis <Scr(2000)> belegt sind, verwenden wir für das Retten eines weiteren Untergrundes die Variablen ab <Scr(2001)>:

```
Call Ptr86(VarSeg, VarOff, VarPtr(Scr(2001)))
Call Windows(Col, Row, Breite, Laenge, VarSeg, VarOff, Flag)
```

In diesem Beispiel ermittelt PTR86 die Adresse der Variablen <Scr(2001)>. Im folgenden Aufruf von WINDOWS wird diese Adresse übergeben. WINDOWS speichert den übergebenen Ausschnitt diesmal in <Scr(2001)>, <Scr(2002)> und so weiter.

```
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    i┌──────────────────────────────┐own-Menüs
Dies    i│  Dies ist das erste Window.  │own-Menüs
Dies    i│  Dies ist das erste Window.  │own-Menüs
Dies    i│  Dies i┌──────────────────────────────────┐
Dies    i│  Dies i│  Dies ist das zweite Window.     │
Dies    i└────────│  Dies ist das zweite Window.     │
Dies    ist    ein│  Dies ist das zweite Window.     │
Dies    ist    ein│  Dies ist das zweite Window.     │
Dies    ist    ein└──────────────────────────────────┘
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
```

Nett, diese Bildschirmdarstellung, nicht wahr? Genau dieses Demo (allerdings optisch weitaus anspruchsvoller!) wird am Ende des Kapitels ein Demoprogramm auf den Bildschirm zaubern. Der »Hintergrund« wird von einem Window überlagert und dieses Window wiederum teilweise von einem zweiten Window verdeckt. Um die Inhalte der betreffenden Bildschirmausschnitte zu retten, gehen wir so vor:

1. Zuerst retten wir den Untergrund unter Window Nummer 1. Nehmen wir an, der rechteckige Ausschnitt umfaßt 120 Zeichen. Diese 120 Zeichen werden in <Scr(1)> bis <Scr(120)> gespeichert. Nun wird Window Nummer 1 ausgegeben.

2. Bevor das zweite Window ausgegeben wird, retten wir auch dessen Untergrund. Wenn es ebenfalls 120 Zeichen umfaßt, wird <Scr(121)> bis <Scr(240)> für die Speicherung benutzt. Nach diesem »Retten« des Untergrunds geben wir Window Nummer 2 aus.

Der Reihe nach »zugeklappt« werden die Windows, indem der gerettete Untergrund auf den Bildschirm zurückgeschrieben wird. Zum Zuklappen von Window 2 wird der Inhalt von <Scr(121)> bis <Scr(240)> in den betreffenden Bildschirmausschnitt zurückkopiert, zum Zuklappen von Window 2 der Inhalt von <Scr(1)> bis <Scr(120)>.

Mit der Assembler-Routine WINDOWS können wir eine unbeschränkte Anzahl von überlappender Windows nach Belieben »aktivieren« und wieder »deaktivieren«. Die Windowanzahl ist nur durch die Arraygröße begrenzt.

Die Programmierung dieses »Windowing« ist allerdings noch sehr aufwendig. Wir müssen jederzeit darüber informiert sein, bis zu welcher Variablen das Array <Scr(..)> belegt ist und in welchem Bereich sich welches gerettete Window befindet. Das heißt, wir betreiben momentan noch einen beträchtlichen Verwaltungsaufwand »per Hand«.

Diesen Verwaltungsaufwand übernehmen die beiden Prozeduren PUSHSCREEN und PULLSCREEN. Sie automatisieren die Window-Verwaltung. Nehmen wir uns zuerst PUSHSCREEN vor. Diese Prozedur übernimmt das Retten eines Bildschirmausschnitts. Erinnern Sie sich an die in COMDEF.BAS deklarierten globalen Variablen <ScrTab(..)> und <ScrPage>.

<ScrPage> gibt an, wie viele Ausschnitte sich im Array <Scr(..)> befinden. Um genau zu sein: <ScrPage> ist gleich der Anzahl geretteter Ausschnitte plus eins.

<ScrTab(..)> ist eine Tabelle, die <ScrPage> Einträge enthält. Jeder Eintrag gibt den Index der ersten Variablen an, ab der ein Ausschnitt gespeichert ist. Der letzte Eintrag gibt an, ab welcher Variablen ein neuer Ausschnitt gespeichert würde.

Nehmen wir das vorige Beispiel, in dem zwei Ausschnitte mit je 120 Zeichen gerettet wurden:

```
ScrTab(1) = 1          'Ausschnitt 1 beginnt ab <Scr(1)>
ScrTab(2) = 121        'Ausschnitt 2 beginnt ab <Scr(121)>
ScrTab(3) = 241        'Ausschnitt 3 beginnt ab <Scr(241)>
```

Zur Interpretation: Der erste gerettete Ausschnitt beginnt ab <Scr(1)>, der zweite Ausschnitt ist ab <Scr(121)> gespeichert. Ein weiterer Ausschnitt würde ab <Scr(241)> gespeichert, der letzten momentan belegten Variablen plus eins.

In INIT.BAS wird <ScrPage> und der erste Tabelleneintrag <ScrTab(1)> initialisiert.

```
ScrPage   = 1      'Aktuelle Pufferseite
ScrTab(1) = 1      'Ersten Tabelleneintrag initialisieren
```

Die Initialisierung von <ScrPage> mit dem Wert 1 heißt soviel wie »als nächster wird der Ausschnitt Nummer 1 gerettet«, also der erste Ausschnitt.

Die Initialisierung von <ScrTab(1)> mit 1 bedeutet, daß dieser erste Ausschnitt ab der ersten Variablen im »Pufferspeicher« <Scr(..)> beginnt, ab <Scr(1)>.

PUSHSCREEN verwaltet die Tabelle und <ScrTab>, die »Nummer des nächsten zu rettenden Ausschnitts«.

```
'       PushScreen
'       ----------
'       Funktion: Rettet Bildschirmausschnitt
'       Hin     : Col/Row : Linke obere Windowecke
'                 Breite  : Windowbreite in Spalten
'                 Laenge  : Windowbreite in Zeilen

Sub PushScreen(Col, Row, Breite, Laenge) Static
  Flag = False
  Call Ptr86(VarSeg, VarOff, VarPtr(Scr(ScrTab(ScrPage))))
  Call Windows(Col, Row, Breite, Laenge, VarSeg, VarOff, Flag)
  ScrTab(ScrPage + 1) = ScrTab(ScrPage) + (Laenge * Breite)
  ScrPage = ScrPage + 1
End Sub
```

<Flag> wird mit <False> initialisiert, da ein Ausschnitt gerettet werden soll. Nach dem Aufruf von PTR86 und WINDOWS befindet sich eine Kopie des Ausschnitts in <Scr(..)>. Um festzustellen, wo er genau gespeichert wurde, müssen wir den komplexen Ausdruck *Scr(ScrTab(ScrPage))* analysieren.

<ScrPage> wurde mit 1 initialisiert. <ScrTab(ScrPage)> können wir daher auch als <ScrTab(1)> schreiben. Da <ScrTab(1)> ebenfalls mit mit 1 initialisiert wurde, ist der Ausdruck *Scr(ScrTab(ScrPage))* gleichbedeutend mit <Scr(1)>. <Scr(1)> ist die Variable, deren Adresse PTR86 ermittelt und ab der WINDOWS den Ausschnitt speichert.

Anschließend wird in der Tabelle vermerkt, ab welcher Variablen ein weiterer zu rettender Ausschnitt gespeichert würde. Der Index dieser Variablen ergibt sich aus der »Startadresse« des aktuellen Ausschnitts plus der Anzahl der belegten Variablen. Da in jeder Variablen genau ein Zeichen (plus Attribut) gespeichert wird, ist die Variablenanzahl identisch mit der Anzahl der im gespeicherten Ausschnitt enthaltenen Zeichen. Die Zeichenanzahl wiederum ergibt sich aus Windowlänge mal Windowbreite.

```
'       PullScreen
'       ----------
'       Funktion: Holt geretteten Bildschirmausschnitt
'       Hin     : Col/Row : Linke obere Windowecke
'                 Breite  : Windowbreite in Spalten
'                 Laenge  : Windowbreite in Zeilen

Sub PullScreen(Col, Row, Breite, Laenge) Static
  Flag = True
  ScrPage = ScrPage - 1
  Call Ptr86(VarSeg, VarOff, VarPtr(Scr(ScrTab(ScrPage))))
  Call Windows(Col, Row, Breite, Laenge, VarSeg, VarOff, Flag)
End Sub
```

7.7 Listing der Pull-down-Menüs (PULLDOWN.BAS)

```
'       ******************************
'       * Windowing + Pull-down-Menüs *
'       ******************************

'       (C) Said Baloui, 1987

REM $INCLUDE: 'comdef.bas'
'-----------------------------------------------------------------------

'       PushScreen
'       ----------
'       Funktion: Rettet Bildschirmausschnitt
'       Hin     : Col/Row : Linke obere Windowecke
'                 Breite  : Windowbreite in Spalten
'                 Laenge  : Windowbreite in Zeilen

Sub PushScreen(Col, Row, Breite, Laenge) Static
  Flag = False
  Call Ptr86(VarSeg, VarOff, VarPtr(Scr(ScrTab(ScrPage))))
  Call Windows(Col, Row, Breite, Laenge, VarSeg, VarOff, Flag)
  ScrTab(ScrPage + 1) = ScrTab(ScrPage) + (Laenge * Breite)
  ScrPage = ScrPage + 1
End Sub
'-----------------------------------------------------------------------

'       PullScreen
'       ----------
'       Funktion: Holt geretteten Bildschirmausschnitt
'       Hin     : Col/Row : Linke obere Windowecke
'                 Breite  : Windowbreite in Spalten
'                 Laenge  : Windowbreite in Zeilen

Sub PullScreen(Col, Row, Breite, Laenge) Static
  Flag = True
  ScrPage = ScrPage - 1
  Call Ptr86(VarSeg, VarOff, VarPtr(Scr(ScrTab(ScrPage))))
  Call Windows(Col, Row, Breite, Laenge, VarSeg, VarOff, Flag)
End Sub
'-----------------------------------------------------------------------

'       InitMenue
'       ---------
'       Funktion: Ermittelt Parameter von aktiviertem Menü
'       Hin     : Menue : Nr. des aktivierten Menüs
'       Zurück  : WCol/WBreite/WLaenge : Ermittelte Menüparameter

Sub InitMenue(Menue, WCol, WBreite, WLaenge, MBefehle$) Static
'       * Get Startspalte *
  Nr = Ø
  WCol = 1
  While Nr < Menue
    WCol = Instr(WCol, Header$, " ")
    Call SearchOther(Header$, " ", WCol)
```

```
    Nr = Nr + 1
  Wend
  WCol = WCol - WOffset

'       * Get Windowbreite/Windowlänge *
  WBreite   = Ø
  WLaenge   = 2
  MBefehle$ = ""
  j = Ø
  While j <> Len(Menue$(Menue))
    i = j + 1
    MBefehle$ = MBefehle$ + Mid$(Menue$(Menue), i, 1)
    j = Instr(i, Menue$(Menue), "#")
    If j - i > WBreite Then WBreite = j - i
    WLaenge = WLaenge + 1
  Wend
  WBreite = WBreite + 2 * WOffset
  Call UpLowCase(MBefehle$)
End Sub
'-----------------------------------------------------------------------

'       GetName
'       -------
'       Funktion: Übergibt einzelnen Namen aus Menüheader
'       Hin     : StringNr : Nummer des gesuchten Strings
'       Zurück  : S$       : String
'                 SCol     : Startspalte des Strings

Sub GetName(StringNr, S$, SCol) Static
  Nr   = Ø
  SCol = 1
  While Nr <> StringNr
    SCol = Instr(SCol, Header$, " ")
    Call SearchOther(Header$, " ", SCol)
    If SCol = Ø Then Exit Sub
    Nr = Nr + 1
  Wend
  ECol  = Instr(SCol, Header$, " ")
  S$    = Mid$(Header$, SCol, ECol - SCol)
End Sub
'-----------------------------------------------------------------------

'       GetString
'       ---------
'       Funktion: Übergibt einzelnen String des angegebenen Menüs
'       Hin     : Menue    : Nr. des aktiven Menüs
'                 StringNr : Nr. des gesuchten Menüstrings
'       Zurück  : S$       : String

Sub GetString(Menue, StringNr, S$) Static
  Nr = 1
  i  = Ø
  While Nr <> StringNr
    Nr = Nr + 1
    i  = Instr(i + 1, Menue$(Menue), "#")
  Wend
  j  = Instr(i + 1, Menue$(Menue), "#")
  S$ = Mid$(Menue$(Menue), i + 1, (j - 1) - i)
```

```
End Sub
'-----------------------------------------------------------------------

'       Rahmen zeichnen
'       ---------------

'       Funktion: Legt Rahmen um übergebenes Rechteck
'       Hin     : WCol/WRow/WBreite/WLaenge : Rechteck-Parameter
'                 Attribut : Darstellungsart
'                 Typ : 1-4, je nach Rahmentyp

'                       1:                  2:
'
'

'                       3:                  4:
'
'

Sub Rahmen(WCol, WRow, WBreite, WLaenge, Attribut, Typ) Static

'       * Rahmenzeichen *
  Dim Rahmen$(4)
  Rahmen$(1) = "╓╜─╢"
  Rahmen$(2) = "╒╛═│"
  Rahmen$(3) = "╓╜─│"
  Rahmen$(4) = "╒╛═╢"

'       * Oberste Zeile *
  S$ = Left$(Rahmen$(Typ), 1) +_
    String$(WBreite - 2, Mid$(Rahmen$(Typ), 5, 1)) +_
    Mid$(Rahmen$(Typ), 2, 1)
  Call PrintF(WCol, WRow, S$, Attribut)

'        * Mittlere Zeilen *
  S$ = Right$(Rahmen$(Typ),1) + Space$(WBreite-2) + Right$(Rahmen$(Typ), 1)
  For i=1 to WLaenge - 2
    Call PrintF(WCol, WRow + i, S$, Attribut)
  Next i

'       * Unterste Zeile *
  S$ = Mid$(Rahmen$(Typ), 3, 1) +_
    String$(WBreite - 2, Mid$(Rahmen$(Typ), 5, 1)) +_
    Mid$(Rahmen$(Typ), 4, 1)
  Call PrintF(WCol, WRow + i, S$, Attribut)
End Sub
'-----------------------------------------------------------------------

'       PrintMenue
'       ----------
'       Funktion: Malt Menü
'       Hin     : Menue  : Nr. des aktiven Menüs
'                 WCol/WBreite/WLaenge : Menüparameter

Sub PrintMenue(Menue, WCol, WBreite, WLaenge) Static

'       * Rahmen zeichnen *
Call Rahmen(WCol, 2, WBreite, WLaenge, Intensiv, 1)
```

```
'       * Window-Innenzeilen *
  Trenn$ = "├" + String$(WBreite - 2, "-") + "┤"
  For i=1 to WLaenge - 2
    Call GetString(Menue, i, S$)
    If S$ <> "-" Then
      Call PrintF(WCol + WOffset, i + 2, S$, Intensiv)
    Else
      Call PrintF(WCol, i + 2, Trenn$, Intensiv)
    End If
 Next i
End Sub
'-----------------------------------------------------------------------

'       BefehlNeu
'       ---------
'       Funktion: - selektierten Befehl normalisieren
'                 - neu selektierten Befehl invertieren
'       Hin     : Menue     : Nr. des aktiven Menüs
'                 CrsRowAlt : Bisherige Cursorzeile im aktiven Menü
'                 CrsRow    : Voraussichtl.neue Cursorzeile im aktiven Menü
'                 WCol      : Startspalte des Menüs
'                 WBreite   : Breite des Menüs


Sub BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite) Static

'       * Alte Cursorzeile normalisieren *
  Call GetString(Menue, CrsRowAlt, S$)
  S$ = Space$(WOffset - 1) + S$ + Space$(WOffset - 1)
  If Len(S$) < WBreite - 2 Then S$ = S$ + Space$((WBreite - 2) - Len(S$))
  Call PrintF(WCol + 1, 2 + CrsRowAlt, S$, Intensiv)

'       * Neue Cursorzeile invertieren *
  Call GetString(Menue, CrsRow, S$)
  S$ = Space$(WOffset - 1) + S$ + Space$(WOffset - 1)
  If Len(S$) < WBreite - 2 Then S$ = S$ + Space$((WBreite - 2) - Len(S$))
  Call PrintF(WCol + 1, 2 + CrsRow, S$, Invers)
End Sub
'-----------------------------------------------------------------------

'       NameNeu
'       -------
'       Funktion: - invertiert aktuellen Menünamen
'                 - normalisiert neuen Menünamen
'       Hin     : MenueAlt : Nr. des bisher aktiven Menüs
'                 Menue    : Nr. des neuen aktiven Menüs


Sub NameNeu(MenueAlt, Menue) Static

'       * Aktuellen Menünamen invertieren *
  Call GetName(MenueAlt, S$, SCol)
  S$ = " " + S$ + " "
  Call PrintF(SCol - 1, 1, S$, Invers)
```

```
'       * Neuen Menünamen invertieren *
  Call GetName(Menue, S$, SCol)
  S$ = " " + S$ + " "
  Call PrintF(SCol - 1, 1, S$, Normal)
End Sub
'-----------------------------------------------------------------------

'       MenueNeu
'       --------
'       Funktion: Behandelt Anwahl eines neuen Menüs
'       Hin     : MenueAlt  : Nr. des bisher aktiven Menüs
'                 Menue     : Nr. des neuen aktiven Menüs
'                 CrsRow    : Nr. der bisher selektierten Menüzeile
'                 WCol/WBreite/WLaenge : Parameter des alten Menüs
'                 MBefehle$ : Anfangsbuchstaben (klein) der Menübefehle
'       Zurück  : CrsRow    : Nr. der neuen selektierten Menü-
                              zeile (immer 1)
'                 WCol/WBreite/WLaenge : Parameter des neuen Menüs


Sub MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge, MBefehle$)_
Static

'       * Alten Untergrund holen *
  Call PullScreen(WCol, 2, WBreite, WLaenge)

'       * Hervorheben eines anderen Menünamens *
  Call NameNeu(MenueAlt, Menue)

'       * Neue Menüparameter holen und Untergrund retten *
  Call InitMenue(Menue, WCol, WBreite, WLaenge, MBefehle$)
  Call PushScreen(WCol, 2, WBreite, WLaenge)
  Call PrintMenue(Menue, WCol, WBreite, WLaenge)

'       * Befehl 1 selektieren *
  CrsRowAlt = 1: CrsRow = 1
  Call BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite)
End Sub
'-----------------------------------------------------------------------

'       PullDown
'       --------
'       Funktion: Verwaltet Pull-down-Menüs
'       Zurück  : Menue     : Nr. des aktiven Menüs
'                 CrsRow    : Nr. des selektierten Befehls

Sub PullDown(Menue, CrsRow) Static

'       *** Initialisierungsteil ***

'       * Init Header *
  Call PrintF(1, 1, Header$, Invers)

'       * Menüanzahl und Menünamen ermitteln *
  MName$ = ""
  i = Ø
  SCol = 1
  While SCol <> Ø
```

```
    i = i + 1
    Call GetName(i, S$, SCol)
    MName$ = MName$ + Left$(S$, 1)
  Wend
  MZahl  = i - 1
  MName$ = Left$(MName$, Len(MName$) - 1)

'       * Init Menue 1 *
  Menue = 1
  Call InitMenue(Menue, WCol, WBreite, WLaenge, MBefehle$)
  Call PushScreen(WCol, 2, WBreite, WLaenge)
  Call MenueNeu(Menue, Menue, CrsRow, WCol, WBreite, WLaenge, MBefehle$)


'       *** Hauptschleife ***
  ExitFlag = False
  While ExitFlag = False
    Call Taste(Key$, Escape)
      If Key$ = zReturn$ Then
        ExitFlag = True
      ElseIf Key$ = zRight$ Then
        MenueAlt = Menue
        Menue    = Menue + 1
        If Menue > MZahl Then Menue = 1
        Call MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge,_
        MBefehle$)
      ElseIf Key$ = zLeft$ Then
        MenueAlt = Menue
        Menue    = Menue - 1
        If Menue = Ø Then Menue = MZahl
        Call MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge,_
        MBefehle$)
      ElseIf Instr(MName$, Key$) <> Ø Then
        MenueAlt = Menue
        Menue    = Instr(MName$, Key$)
        Call MenueNeu(MenueAlt, Menue, CrsRow, WCol, WBreite, WLaenge,_
        MBefehle$)
      ElseIf Key$ = zDown$ Then
        CrsRowAlt = CrsRow
        CrsRow    = CrsRow + 1
        If CrsRow > WLaenge - 2 Then CrsRow = 1
        Call GetString(Menue, CrsRow, S$)
        If S$ = "-" Then CrsRow = CrsRow + 1
        Call BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite)
      ElseIf Key$ = zUp$ Then
        CrsRowAlt = CrsRow
        CrsRow    = CrsRow - 1
        If CrsRow = Ø Then CrsRow = WLaenge - 2
        Call GetString(Menue, CrsRow, S$)
        If S$ = "-" Then CrsRow = CrsRow - 1
        Call BefehlNeu(Menue, CrsRowAlt, CrsRow, WCol, WBreite)
      ElseIf Instr(MBefehle$, Key$) <> Ø Then
        CrsRow   = Instr(MBefehle$, Key$)
        ExitFlag = True
      End If
  Wend
```

```
'       * Selektierten Befehl ermitteln *
  Call PrintF(1, 1, Header$, Invers)
  Call PullScreen(WCol, 2, WBreite, WLaenge)
  For i=1 to CrsRow
    Call GetString(Menue, i, S$)
    If S$ = "-" Then CrsRow = CrsRow - 1
  Next i
End Sub
```

7.8 Einbindung in die User-Library

Die Einbindung in die User-Library ist nicht ganz so einfach, wie Sie vielleicht glauben. PULLDOWN enthält bereits die Anweisung COMDEF.BAS zur Eingliederung der Deklarations-Datei. Theoretisch sollte es ausreichen, PULLDOWN mit unserem »Kompilier-Batch« zu bearbeiten und anschließend zusammen mit INPUT, ROUTINEN und MASKE in die User-Library einzubinden.

Nun, was das Kompilieren betrifft, haben Sie recht. Also verlassen Sie QuickBASIC, kopieren Sie PULLDOWN.BAS auf die Arbeitsdiskette (Festplatte: in das Arbeitsverzeichnis) und geben Sie ein:

Diskette:	Festplatte:
`A>DC PULLDOWN`	`C>PC MASKE`

Auf der Programmdiskette (Platte: im Verzeichnis SYSTEM20) befindet sich nun außer ROUTINEN.OBJ, INPUT.OBJ und MASKE.OBJ zusätzlich die Objektdatei PULLDOWN.OBJ.

Und jetzt wird's problematisch. Es genügt bei weitem nicht, diese vier Objektdateien zu einer User-Library »zusammenzulinken«!

PULLDOWN verwendet drei Assembler-Routinen:

- PRINTF: schnelle Bildschirmausgabe
- WINDOWS: retten/holen eines Bildschirmausschnitts
- PTR86: Ermitteln der Speicheradresse einer Variablen

PTR86 ist keine Routine von mir, sondern gehört zum QuickBASIC-System. Wenn Assembler-Routinen verwendet werden, sollte zusätzlich die Datei PREFIX.OBJ eingebunden werden, die ebenfalls zum QuickBASIC-System gehört.

Da ich mir die Freiheit nahm, beide Routinen auf die Diskette zum Buch zu kopieren, finden Sie darauf alle vier benötigten Files als »einbindungsfähige« Objektdateien. Achtung: Die Datei, in der sich die Prozedur PTR86 befindet, heißt nicht PTR86.OBJ, sondern INT86.OBJ!

Kopieren Sie folgende Objektdateien auf unsere Programmdiskette:

PREFIX.OBJ
INT86.OBJ
PRINTF.OBJ
WINDOWS.OBJ

Auf unserer Programmdiskette befinden sich außer diesen vier Dateien die bereits vorhandenen Objektdateien ROUTINEN.OBJ, INPUT.OBJ und MASKE.OBJ.

Wir erstellen nun eine User-Library, die alle (!) unsere Prozeduren enthält:

Diskette:

```
A>DL ROUTINEN INPUT MASKE PREFIX INT86 PRINTF WINDOWS PULLDOWN
```

Festplatte:

```
A>PL ROUTINEN INPUT MASKE PREFIX INT86 PRINTF WINDOWS PULLDOWN
```

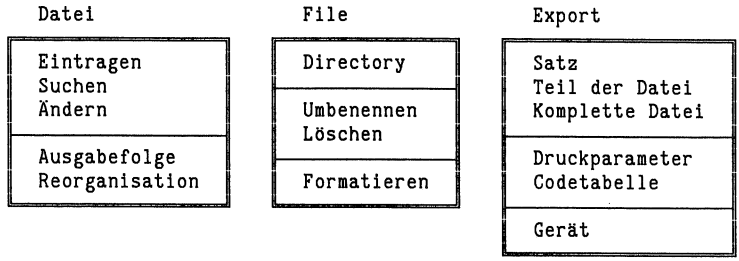
Ein wahnsinnig langer Aufruf, nicht wahr? Als »Gegenleistung« erhalten wir eine rund 23 Kbyte große User-Library, die circa 650 BASIC- und 600 Assembler-Programmzeilen in ausführbarer Form enthält!

Diese recht umfangreiche Library können wir ohne jede Kompilieraktion in beliebigen Programmen einsetzen. Bevor wir ein solches »integrierendes« Programm erstellen, zeige ich Ihnen an zwei kleinen Demoprogrammen, wie unser Projekt PULLDOWN effektiv eingesetzt wird.

7.9 Aufruf der Pull-down-Menüs (PULLDEMO.BAS)

Die Prozedur PULLDOWN ist derart »mächtig«, daß zur Demonstration ein vergleichsweise winziges Programm genügt. Sie finden es unter dem Namen PULLDEMO im Verzeichnis DEMOS.

Das Programm gibt ein Pull-down-Menü mit drei einzelnen Menüs aus:



Am Anfang dieses Hauptprogramms werden wie üblich mit $INCLUDE die Deklarations- und die Initialisierungs-Datei eingebunden. Die globalen Variablen sind nun deklariert und mit den benötigten Werten initialisiert.

```
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
'-----------------------------------------------------------------------

Header$  = "      Datei  File  Export
"
Menue$(1) = "Eintragen/Suchen/Editieren/-/Ausgabefolge/Reorganisation/"
Menue$(2) = "Directory/-/Umbenennen/Löschen/-/Formatieren/"
Menue$(3) = "Satz/Teil der Datei/Komplette Datei/-
/Druckparameter/Codetabelle/-/Gerät/"

For i=1 to 20
  Print "   Dies    ist   ein      Demo    der      Pull-Down-Menüs"
Next i

Call PullDown(Menue, Befehl)
Print Menue, Befehl
End
```

<Header$> enthält die Menüleiste mit drei Menünamen. Denken Sie daran: Vor dem ersten Menünamen müssen sich mindestens zwei Leerzeichen befinden! Hier werden noch mehr Leerzeichen verwendet, damit sich das erste Menü nicht unmittelbar am linken Bildschirmrand befindet. Ein wenig Randabstand verbessert die Optik (Geschmackssache, Änderungen stehen Ihnen frei).

<Header$> ist genau 80 Zeichen lang, damit bei der inversen Ausgabe nicht nur ein Teil, sondern die gesamte oberste Bildschirmzeile einheitlich invers ist.

<Menue$(1)> bis <Menue$(3)> enthalten die drei Menüs (Kommandos und Trennstriche).

In einer Schleife wird 20mal die Zeichenkette » Dies ist ein Demo der Pull-down-Menüs« ausgegeben. Auf diese Weise wird der Bildschirm gefüllt und Sie sehen später beim Blättern in den Menüs, daß der ursprüngliche Untergrund tatsächlich wiederhergestellt wird.

Beim Aufruf von PULLDOWN werden nur zwei Variablen angegeben. Per Referenz, da PULLDOWN in diesen Variablen die Nummer des zuletzt aktiven Menüs und des darin gewählten Kommandos übergibt.

Regeln für die Definition eines Pull-down-Menüs

1. Der Leistenstring (<Header$>) beginnt mit mindestens zwei Leerzeichen. Zwischen den Menünamen befindet sich mindestens ein Leerzeichen. Alle Menünamen müssen mit unterschiedlichen Anfangsbuchstaben beginnen, da sonst keine Direktanwahl möglich ist.
2. Zu jedem Menünamen gehört ein in dem globalen Array <Menue$(..)> definierter Menüstring (<Menue$(1)>, <Menue$(2)> etc.)
3. Menüs bestehen aus Kommandos und optionalen Trennlinien zwischen einzelnen Kommandoblöcken. Trennlinien werden mit einem Bindestrich definiert. Jedes Kommando – und auch ein Bindestrich! – endet mit dem Zeichen »#«. Alle Kommandos eines Menüs müssen mit unterschiedlichen Anfangsbuchstaben beginnen (Direktanwahl!).
4. Beim Aufruf werden PULLDOWN zwei numerische Variablen per Referenz übergeben. Die erste enthält nach der Rückkehr die Nummer des aktiven Menüs, die zweite die Nummer des gewählten Kommandos.

Bedienung der Pull-down-Menüs

1. Auf- und Zuklappen von Menüs mit CURSOR RECHTS/CURSOR LINKS oder direkt über SHIFT+Anfangsbuchstabe des gewünschten Menüs.
2. Selektion eines Kommandos mit CURSOR OBEN/CURSOR UNTEN.
3. Anwählen eines Kommandos mit RETURN (selektiertes Kommando) oder direkt über den Anfangsbuchstaben.

7.10 Aufruf der Window-Routinen

WINDOWS.BAS (Verzeichnis DEMOS) ist ein Demoprogramm, das die weit über Pull-down-Menüs hinausgehenden Möglichkeiten der Window-Verwaltung zeigt.

Sie wissen inzwischen, daß es möglich ist, mit PULLSCREEN und PUSHSCREEN sogar einander überlappende Windows zu verwalten. Das Prinzip:

```
Window Nummer 1 retten
Window Nummer 2 retten
...
...
Window Nummer N retten

Window Nummer N holen
...
...
Window Nummer 2 holen
Window Nummer 1 holen
```

Die Reihenfolge beim »Zuklappen« der Windows ist der Reihenfolge beim »Aufklappen« genau entgegengesetzt.

Da bei leerem Untergrund die Wirkungsweise von WINDOWS.BAS nicht sichtbar wird, füllt das Programm den Bildschirm mit einem Demotext.

```
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
'-----------------------------------------------------------------------

'       * Demountergrund *
For i=1 to 25
  Print "Dies    ist    ein    Demo    der    Pull-down-Menüs"
Next i
```

Das Programm wartet darauf, daß Sie eine beliebige Taste drücken und klappt anschließend das erste Window auf.

```
'       * Window 1 auf *
Label1: A$ = Inkey$: If A$ = "" Then Goto Label1
Col1 = 10: Row1 = 10: Breite1 = 34: Laenge1 = 6
Call PushScreen(Col1, Row1, Breite1, Laenge1)
Color 7, 0
Locate Row1,   Col1: Print "┌──────────────────────────────┐"
Locate Row1+1, Col1: Print "│  Dies ist das erste Window.  │"
Locate Row1+2, Col1: Print "│  Dies ist das erste Window.  │"
Locate Row1+3, Col1: Print "│  Dies ist das erste Window.  │"
Locate Row1+4, Col1: Print "│  Dies ist das erste Window.  │"
Locate Row1+5, Col1: Print "└──────────────────────────────┘"
```

Ein Window ist immer rechteckig. Die Rechteck-Parameter werden definiert und PUSHSCREEN übergeben, um den momentanen Bildschirminhalt in diesem Ausschnitt zu retten. Danach wird das Window ausgegeben.

```
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    i┌──────────────────────────────────────┐own-Menüs
Dies    i│  Dies ist das erste Window.          │own-Menüs
Dies    i│  Dies ist das erste Window.          │own-Menüs
Dies    i│  Dies ist das erste Window.          │own-Menüs
Dies    i│  Dies ist das erste Window.          │own-Menüs
Dies    i└──────────────────────────────────────┘own-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
```

Wieder wartet das Programm auf eine Taste und klappt danach das zweite Window auf.

```
'        * Window 2 auf *
Label2: A$ = Inkey$: If A$ = "" Then Goto Label2
Col2 = 20: Row2 = 13: Breite2 = 34: Laenge2 = 6
Call PushScreen(Col2, Row2, Breite2, Laenge2)
Color 0, 7
Locate Row2,   Col2: Print "┌──────────────────────────────────┐"
Locate Row2+1, Col2: Print "│  Dies ist das zweite Window.     │"
Locate Row2+2, Col2: Print "│  Dies ist das zweite Window.     │"
Locate Row2+3, Col2: Print "│  Dies ist das zweite Window.     │"
Locate Row2+4, Col2: Print "│  Dies ist das zweite Window.     │"
Locate Row2+5, Col2: Print "└──────────────────────────────────┘"
```

Erneut werden die Rechteck-Parameter definiert. Beide Windows sind gleich breit und lang. Da sich die beiden Windows überlappen sollen, wird die obere linke Ecke etwas rechts und unterhalb der oberen linken Ecke von Window 1 definiert.

Der Inhalt des betreffenden Ausschnitts wird mit PUSHSCREEN gerettet und das Window ausgegeben.

```
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    i                            own-Menüs
Dies    i   Dies ist das erste Window.   own-Menüs
Dies    i   Dies ist das erste Window.   own-Menüs
Dies    i   Dies i
Dies    i   Dies i   Dies ist das zweite Window.
Dies    i            Dies ist das zweite Window.
Dies    ist    ein   Dies ist das zweite Window.
Dies    ist    ein   Dies ist das zweite Window.
Dies    ist    ein
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
Dies    ist    ein    Demo    der    Pull-Down-Menüs
```

Obwohl es in der Abbildung nicht sichtbar ist: Das zweite Window wird invers ausgegeben (*Color 0, 7*).

Beide Windows werden der Reihe nach »zugeklappt«. Das Programm wartet auf eine Taste und klappt zuerst Window Nummer 2 zu (umgekehrte Reihenfolge wie beim Aufklappen).

```
'        * Window 2 zu *
Label3: A$ = Inkey$: If A$ = "" Then Goto Label3
Call PullScreen(Col2, Row2, Breite2, Laenge2)
```

Zuklappen heißt, daß PUSHSCREEN mit den Parametern des zweiten Windows aufgerufen wird, um den Inhalt dieses Bildschirmausschnitts vor dem Aufklappen wiederherzustellen.

Auf die gleiche Weise wird nach Betätigung einer Taste das erste Window zugeklappt.

```
'        * Window 1 zu *
Label4: A$ = Inkey$: If A$ = "" Then Goto Label4
Call PullScreen(Col1, Row1, Breite1, Laenge1)
```

Auf diese Routine bin ich ein klein wenig stolz (merkt man's?). PUSHSCREEN/PULLSCREEN verwalten Windows mit Hilfe der Assembler-Routine WINDOWS so optimal, daß der Programmierer fast schon arbeits-

los ist. Daher will ich dieses Kapitel nicht abschließen, ohne Ihnen einige Ideen für die Anwendung mitzugeben.

1. Einblendung von Hilfstexten: Blenden Sie auf Tastendruck Hilfstexte in Ihre Programme ein. Die übliche Methode, für Hilfstexte immer den gesamten Bildschirm zu benutzen, ist ja ganz nett, aber in einem Window, das den Originalbildschirm nur teilweise überlagert, wirken Hilfstexte weit ansprechender.
2. Wickeln Sie Standard-Dialoge (»Datei löschen (j/n) ? n«) ähnlich wie QuickBASIC in einem Window ab.
3. Geben Sie die Directory in einem Window aus, am besten in einem »scrollenden« Window, dessen Inhalt der Benutzer nach oben/unten verschieben kann.

Es ist nur eine Frage der Phantasie, welche Anwendungen Ihnen noch einfallen. Und überlegen Sie sich, bei welcher Anwendung überlappende Windows zu gebrauchen sind. Es wirkt optisch sehr attraktiv, wenn ein Window von einem weiteren überlagert wird (in dem zum Beispiel eine Fehlermeldung erscheint?).

7.11 Listing von WINDOWS.BAS

```
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
'-----------------------------------------------------------------------

'       * Demountergrund *
For i=1 to 25
  Print "Dies    ist    ein    Demo    der    Pull-down-Menüs"
Next i

'       * Window 1 auf *
Label1: A$ = Inkey$: If A$ = "" Then Goto Label1
Col1 = 10: Row1 = 10: Breite1 = 34: Laenge1 = 6
Call PushScreen(Col1, Row1, Breite1, Laenge1)
Color 7, 0
Locate Row1,   Col1: Print "                                    "
Locate Row1+1, Col1: Print "  Dies ist das erste Window.        "
Locate Row1+2, Col1: Print "  Dies ist das erste Window.        "
Locate Row1+3, Col1: Print "  Dies ist das erste Window.        "
Locate Row1+4, Col1: Print "  Dies ist das erste Window.        "
Locate Row1+5, Col1: Print "                                    "

'       * Window 2 auf *
Label2: A$ = Inkey$: If A$ = "" Then Goto Label2
Col2 = 20: Row2 = 13: Breite2 = 34: Laenge2 = 6
Call PushScreen(Col2, Row2, Breite2, Laenge2)
```

```
Color Ø, 7
Locate Row2,   Col2: Print "╔═══════════════════════════╗"
Locate Row2+1, Col2: Print "│  Dies ist das zweite Window.  │"
Locate Row2+2, Col2: Print "│  Dies ist das zweite Window.  │"
Locate Row2+3, Col2: Print "│  Dies ist das zweite Window.  │"
Locate Row2+4, Col2: Print "│  Dies ist das zweite Window.  │"
Locate Row2+5, Col2: Print "╚═══════════════════════════╝"

'       * Window 2 zu *
Label3: A$ = Inkey$: If A$ = "" Then Goto Label3
Call PullScreen(Col2, Row2, Breite2, Laenge2)

'       * Window 1 zu *
Label4: A$ = Inkey$: If A$ = "" Then Goto Label4
Call PullScreen(Col1, Row1, Breite1, Laenge1)
```

8

Konvertieren von GW-BASIC-Programmen

Wir erstellen nun das Programm CONVERT, das sich wie üblich unter dem Namen CONVERT.BAS im Verzeichnis MODULE befindet. Außer der eigentlichen Anwendung, der »Konvertierung« Ihrer GW-BASIC-Programme, soll Ihnen dieses Programm zeigen, wie einige neue »Features« von QuickBASIC in der Praxis eingesetzt werden:

1. Wie Sie ein »Stand-alone«-Programm erzeugen, das auch ohne QuickBASIC lauffähig ist.
2. Welche Möglichkeiten dynamische Arrays und die Anweisungen $INCLUDE, REDIM und ERASE bieten.
3. Wann der Einsatz »statischer« Variablen vorteilhaft ist.
4. Globale Variablen integrierten wir bisher per $INCLUDE in all unsere Programmodule. Manchmal benötigt man zusätzlich Variablen, die nur für ein einziges Programm global sein sollen. Die gemeinsame Benutzung solcher »unterschiedlicher« globaler Variablen bietet eine Vielzahl an Fehlerquellen. CONVERT zeigt Ihnen, wie in der Praxis vorzugehen ist.
5. Das Programm enthält eine Fehlerbehandlungsroutine. Nun ist die Fehlerbehandlung nichts Neues. Auch GW-BASIC besitzt entsprechende Anweisungen. Bei QuickBASIC sind jedoch einige Besonderheiten zu berücksichtigen, zum Beispiel die Einstellung der benötigten Kompilieroptionen.
6. Wie unsere User-Library effektiv eingesetzt wird, vor allem die Pulldown-Menüs, die Windowing-Routinen und die Eingaberoutine.

Sie sehen, CONVERT enthält so ziemlich alles, was QuickBASIC an Neuheiten bietet.

Zu Punkt 1: Bisher erstellten wir immer neue Module für die User-Library. Irgendwann möchte man jedoch auch einmal – unter Benutzung der User-Library – ein »Anwendungsprogramm« schreiben.

Wenn ein Programm endgültig ausgetestet ist, können Sie mit Hilfe einer neuen Kompilieroption und des Linkers LINK.EXE ein EXE-Programm erzeugen. Dieses Programm beinhaltet außer dem Hauptprogramm auch die User-Library und alle benötigten Routinen des QuickBASIC-Systems (Routinen zur Ausführung von PRINT, INPUT etc.).

Da alle benötigten Teile des QuickBASIC-Systems in der erzeugten Programmdatei enthalten sind, wird QuickBASIC selbst zur Ausführung nicht mehr benötigt. Das Programm ist allein lauffähig.

Ohne die Möglichkeit, derartige »Stand-alone-Programme« zu erzeugen, können Sie theoretisch niemals QuickBASIC-Programme weitergeben. Denn das QuickBASIC-System dürfen Sie natürlich nicht einfach »schnell mal kopieren«.

Aber selbst wenn nur Sie Ihre Programme benutzen, ist diese Option interessant. Es ist ärgerlich, zur Benutzung eines ausgereiften Programms immer wieder QuickBASIC (und die User-Library) sowie das Hauptprogramm zu laden und es zu kompilieren.

CONVERT wird zuerst auf dem üblichen Weg entwickelt. Anschließend wird daraus ein allein lauffähiges Programm erzeugt. CONVERT verwendet einige Eigenschaften von QuickBASIC, die wir bisher noch nicht nutzten: dynamische Arrays, statische Variablen und die Fehlerbehandlung, über die auch schon GW-BASIC verfügt.

Ein Bitte: CONVERT ist ein sehr komplexes Programm. Ich kann Sie verstehen, wenn Sie irgendwann die Lust zum Weiterlesen verlieren. Sie werden viel Ausdauer benötigen, um sich in dieses Programm einzuarbeiten. Ich selbst werde es in einigen Monaten bestimmt nicht mehr in allen Einzelheiten verstehen.

Aber auch wenn Sie irgendwann die Lust verlieren: Zumindest »anlesen« sollten Sie auf alle Fälle folgende Kapitel:

- »Variablen-Deklaration und -Definition« (die Nutzung globaler Variablen für alle(!) Programmodule zusammen mit Variablen, die nur für ein(!) Modul global sind).
- Den Beginn von »Laden eines GW-BASIC-Programms«, der zeigt, worauf Sie bei der Verwendung dynamischer Arrays achten müssen.
- »Aktuelles Verzeichnis wechseln« verwendet zum ersten Mal in diesem Buch wirklich sinnvoll statische Variablen.

8.1 Features von CONVERT

CONVERT konvertiert GW-BASIC-Programme zur Weiterbearbeitung mit QuickBASIC. »Konvertieren« ist allerdings nicht ganz der richtige Ausdruck. Eine der netten Eigenschaften von QuickBASIC ist ja gerade die nahezu perfekte »Aufwärtskompatibilität« zu GW-BASIC. Nahezu jedes GW-BASIC-Programm können Sie mit QuickBASIC problemlos kompilieren und um ein Vielfaches schneller als zuvor ablaufen lassen.

Aber: An QuickBASIC kann man sich schnell gewöhnen. So schnell, daß Programme mit Zeilennummern und Befehlen wie GOTO 1234 nach kurzer Zeit geradezu »anachronistisch« erscheinen.

Vielleicht geht es Ihnen inzwischen wie mir. Für dieses Buch konnte ich ein Programm gut gebrauchen, das ich vor kurzem in GW-BASIC schrieb. Ich wollte aus dem Originalprogramm die Zeilennummern entfernen und für die Sprungbefehle Label verwenden. Ein Auszug aus dem Programm:

```
Originalprogramm

.........
.........
.........
820   X=INSTR(J,PZ$(I),"THEN"): REM 'then'  ab <j> vorhanden?
830   Y=INSTR(J,PZ$(I),"GOTO"): REM 'goto'  ab <j> vorhanden?
840   Z=INSTR(J,PZ$(I),"GOSUB"):REM 'gosub' ab <j> vorhanden?
850   IF X=0 AND Y=0 AND Z=0 THEN GOTO 980:REM kein sprungbefehl
860   :
870   GOSUB 470:REM bis zu erstem space vortasten
880   GOSUB 540:REM folgende spaces überlesen
890   :
900   IF J=Y OR J=Z THEN GOTO 870:REM 'then goto'/'then gosub'
910   IF INSTR(ZIFF$,MID$(PZ$(I),J,1))=0 THEN GOTO 820
920   :
930   GOSUB 610:ZN$(NR)=ZN$:REM zeilennummer lesen
940   :
950   NR=NR+1
960   IF J=LEN(PZ$(I)) THEN GOTO 980
970   IF MID$(PZ$(I),J,1)="," THEN J=J+1:GOTO 930 ELSE GOTO 820
980 NEXT I
990 RETURN
.........
.........
.........
```

Das Resultat meiner Änderungen sollte so aussehen:

```
.........
.........
.........
NextJump: X=INSTR(J,PZ$(I),"THEN"): REM 'then'  ab <j> vorhanden?
  Y=INSTR(J,PZ$(I),"GOTO"): REM 'goto'  ab <j> vorhanden?
  Z=INSTR(J,PZ$(I),"GOSUB"):REM 'gosub' ab <j> vorhanden?
```

```
  IF X=Ø AND Y=Ø AND Z=Ø THEN GOTO NextLine:REM kein sprungbefehl
  :
ReadJumpAgain: GOSUB ReadSpace:REM bis zu erstem space vortasten
  GOSUB ReadNoSpace:REM folgende spaces überlesen
  :
  IF J=Y OR J=Z THEN GOTO ReadJumpAgain:REM 'then goto'/'then gosub'
  IF INSTR(ZIFF$,MID$(PZ$(I),J,1))=Ø THEN GOTO NextJump
  :
NextNumber: GOSUB ReadLineNr:ZN$(NR)=ZN$:REM zeilennummer lesen
  :
  NR=NR+1
  IF J=LEN(PZ$(I)) THEN GOTO NextLine
  IF MID$(PZ$(I),J,1)="," THEN J=J+1:GOTO NextNumber ELSE GOTO NextJump_
  NextLine: NEXT I
RETURN
.........
.........
.........
```

Schon eher »QuickBASIC-like«, finden Sie nicht? Was Sie hier sehen, ist jedoch nur ein kleiner Teil des gesamten Programms. Das ganze Programm entsprechend zu ändern, wäre extrem aufwendig. Zuerst ist zu ermitteln, welche Zeilen angesprungen werden. Genau diese »Zielzeilen« sollen statt mit Zeilennummern mit alphanumerischen Label versehen werden.

Die Ermittlung der Zielzeilen ist in einem größeren Programm sehr aufwendig. Es tröstet wenig, zu wissen, daß der Rest dank der Such- und Ersetzfunktionen eines komfortablen Editors kaum noch Mühe bereitet.

Das Resultat: Statt das Programm zu übernehmen, schrieb ich es komplett neu! CONVERT soll diesen Aufwand vermeiden und Ihre GW-BASIC-Programme in eine »vernünftige« Form bringen.

CONVERT entfernt aus dem angegebenen Programm alle Zeilennummern und setzt statt dessen »Standard-Label« für Sprungangaben ein (»Label1«, »Label2« etc.).

Außerdem stellt Ihnen CONVERT noch einige zusätzliche Optionen zur Verfügung:

1. Sie können jedes der automatisch eingesetzen Standardlabel im gesamten Text durch ein Label Ihrer Wahl ersetzen, zum Beispiel »Label12« durch »BubbleSort«.

2. CONVERT ersetzt auf Wunsch alle REM's durch das in QuickBASIC als Kommentarzeichen zulässige »'«.

3. Die einheitliche Großschrift von GW-BASIC (PRINT, INPUT etc.) kann im gesamten Text durch Groß-Klein-Schrift ersetzt werden (Print, Input etc.).

Der Programmauszug sieht nach Ausnutzung aller Möglichkeiten von CONVERT so aus:

```
.....
.....
.....
NextJump: X=Instr(J,Pz$(I),"Then"): ' 'then'  ab <j> vorhanden?
  Y=Instr(J,Pz$(I),"Goto"): ' 'goto'  ab <j> vorhanden?
  Z=Instr(J,Pz$(I),"Gosub"):' 'gosub' ab <j> vorhanden?
  If X=Ø And Y=Ø And Z=Ø Then Goto NextLine:' kein sprungbefehl
  :
ReadJumpAgain: Gosub ReadSpace:' bis zu erstem space vortasten
  Gosub ReadNoSpace:' folgende spaces überlesen
  :
  If J=Y Or J=Z Then Goto ReadJumpAgain:' 'then goto'/'then gosub'
  If Instr(Ziff$,Mid$(Pz$(I),J,1))=Ø Then Goto NextJump
  :
NextNumber: Gosub ReadLineNr:Zn$(Nr)=Zn$:' zeilennummer lesen
  :
  Nr=Nr+1
  If J=Len(Pz$(I)) Then Goto NextLine
  If Mid$(Pz$(I),J,1)="," Then J=J+1:Goto NextNumber Else Goto NextJump_
  NextLine: Next I
Return
.....
.....
.....
```

Das ist schon eher ein Programm, mit dem man sich anfreunden könnte. Aber glauben Sie mir: Bis CONVERT fertig ist, liegt ein weiter Weg vor Ihnen. Aber das Buch wendet sich ja ausdrücklich an Fortgeschrittene. Und ich werde mich hüten, fortgeschrittene Programmierer mit ein paar Schleifen und ein wenig PRINT oder INPUT als Beigabe zu langweilen.

8.2 Grundlegende Routinen

Bevor wir zu den wirklich anspruchsvollen Teilen vordringen, stelle ich Ihnen zwei »harmlosere«, aber häufig verwendete Routinen vor: WARTEN und STANDINPUT. Beide können Sie bestimmt gut in Ihren eigenen Programmen gebrauchen.

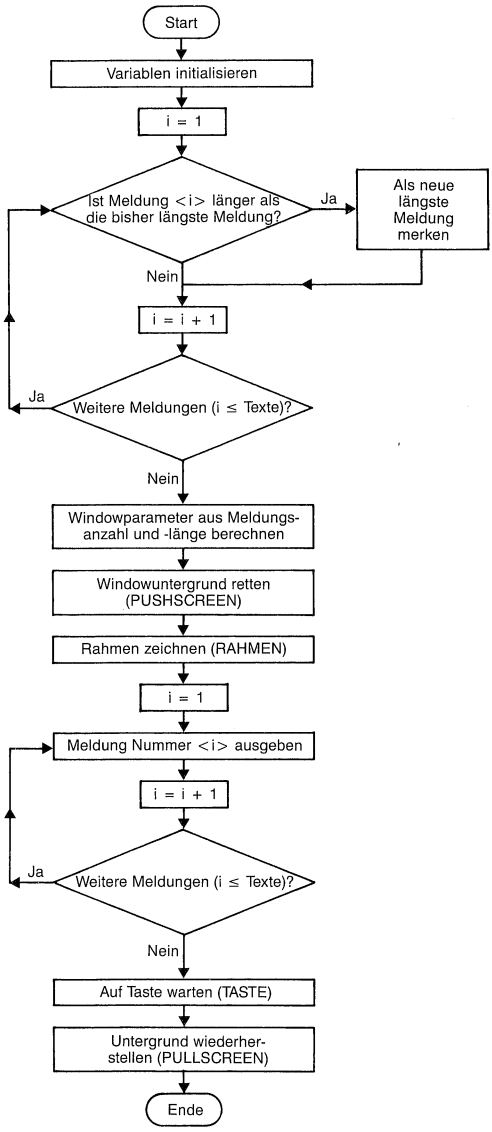


Bild 17 *Meldung ausgeben (WARTEN)*

WARTEN gibt beliebige Meldungen in einem Window aus. Das Window wird »aufgeklappt« und darauf gewartet, daß der Benutzer die Meldung registriert (eine Taste drückt). Anschließend wird das Window »zugeklappt«.

```
'       Meldung ausgeben/Auf Taste warten
'       --------------------------------
'       Funktion: Gibt Meldung in Window aus und wartet auf eine Taste
'       Hin     : Text$(..) : Meldung
'                 Texte     : Anzahl der Meldungen
'                 Attribut  : Darstellungsart (normal, invers etc.)
'       Zurück  : Key$      : Gedrückte Taste
'       Aufbau des Windows  :
'                             ┌──────────────────────────┐
'                             │ Meldung 1 (Text$(1))     │
'                             │ Meldung 2 (Text$(2))     │
'                             │ Meldung 3 (Text$(3))     │
'                             └──────────────────────────┘
```

WARTEN setzt voraus, daß <Text$(..)> global deklariert ist. Diese Deklaration wird ausnahmsweise nicht in COMDEF.BAS vorgenommen, sondern im Hauptprogramm.

In <Text$(..)> speichert das aufrufende Programm die Meldungen und in <Texte> übergibt es die Anzahl der »Meldungsstrings«. Den Parameter <Attribut> kennen Sie inzwischen. Er bestimmt, wie das Window dargestellt wird (Normal, Invers etc.). WARTEN übergibt dem aufrufenden Programm in <Key$> die gedrückte Taste zurück.

Zuerst werden die Parameter des Windows ermittelt, die von der Anzahl und der Länge der Meldungen abhängen.

```
Sub Warten(Texte, Key$, Attribut) Static

'       * Parameter des Gesamtwindows ermitteln *
  WLaenge = 2 + Texte
  WBreite = Ø
    For i=1 to Texte
      If Len(Text$(i)) > WBreite Then WBreite = Len(Text$(i))
    Next i
  WBreite = WBreite + 4
  WCol = 4Ø - Int(WBreite / 2)
  WRow = 26 - WLaenge
```

Die Windowlänge beträgt *Texte* + 2, eine Zeile für jeden Meldungsstring plus zwei Zeilen für die obere und und die untere Rahmenzeile.

Die Windowbreite richtet sich nach der längsten Meldung. In einer Schleife wird der längste String ermittelt und seine Länge in <WBreite> gespeichert.

Anschließend wird <WBreite> um 4 erhöht, um die rechte und linke Rahmenbegrenzung und je ein Leerzeichen rechts beziehungsweise links vom Text zu berücksichtigen.

```
┌──────────────────────────┐
│ So sieht eine Meldung    │
│ aus, die aus drei        │
│ Zeilen besteht.          │
└──────────────────────────┘
```

Die letzte Windowzeile soll sich in Zeile 25 befinden, am unteren Bildschirmrand. Sie wird mit *WRow = 26 – WLaenge* ermittelt.

Um das Window wie in der Abbildung in der Mitte des Bildschirms zu zentrieren, wird die Startspalte mit der Formel *WCol = 40 – Int(WBreite / 2)* berechnet (40 minus der Hälfte der Windowbreite).

PUSHSCREEN rettet den aktuellen Untergrund. Anschließend wird das Window ausgegeben.

```
'       * Untergrund retten/Rahmen ausgeben *
  Call PushScreen(WCol, WRow, WBreite, WLaenge)
  Call Rahmen(WCol, WRow, WBreite, WLaenge, Attribut, 4)
```

Nun wissen Sie, warum PULLDOWN eine derart allgemeine Routine zum Rahmenzeichnen enthält: Um zu vermeiden, daß PULLDOWN eine spezielle Routine für die Ausgabe von Pull-down-Menüs inclusive Rahmen und CONVERT eine allgemeinere zum Zeichnen beliebiger Rahmen enthält.

Um den Rahmen in der gewünschten Darstellung auszugeben, gibt WARTEN den übergebenen Parameter <Attribut> einfach an RAHMEN weiter. Der Rahmentyp ist festgelegt (Typ 4), kann aber jederzeit von Ihnen in der Datei CONVERT.BAS geändert werden.

```
'       * Texte ausgeben *
  For i=1 to Texte
    Call PrintF(WCol + 2, WRow + i, Text$(i), Attribut)
  Next i
```

Die Texte werden ebenfalls in der Darstellungsart <Attribut> ausgegeben und zwar ab Spalte *WCol + 2* (Startspalte plus 2 wegen dem linken Rahmenzeichen und dem Leerzeichen zwischen Rahmen und Text).

Die Ausgabezeile zu ermitteln, ist ein klein wenig schwieriger, da sie nicht konstant ist. <i> ist die Nummer der auszugebenden Meldung. Die Ausgabezeile ist *WRow + i*, also genau <i> Zeilen unterhalb der oberen Rahmenlinie.

Das komplette Window – Rahmen plus Meldungen – ist nun sichtbar. Die Routine wartet auf eine Taste und stellt anschließend mit PULLSCREEN den geretteten Untergrund wieder her.

```
'       * Auf Taste warten/Untergrund holen *
  Call Taste(Key$, Esc)
  Call PullScreen(WCol, WRow, WBreite, WLaenge)
End Sub
```

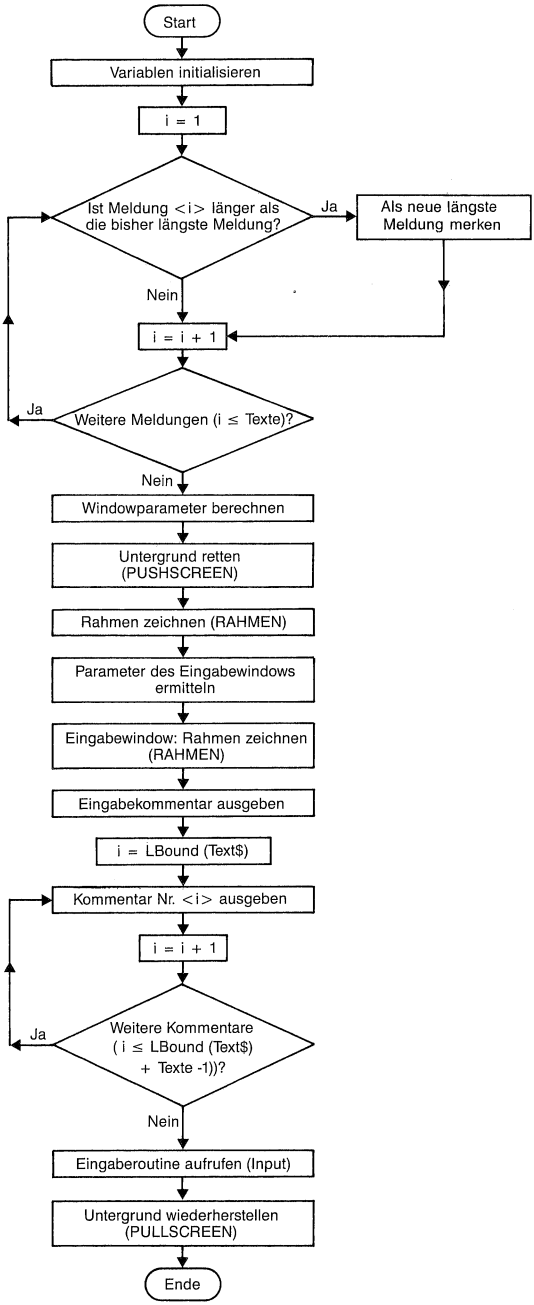


Bild 18 *Standardeingabe behandeln (STANDINPUT)*

Für Eingaben benutzen wir selbstverständlich nicht INPUT, sondern LINPUT, unsere eigene Eingaberoutine. LINPUT allein sorgt zwar für sichere und komfortable Eingaben. Mit den Windowing-Routinen und RAHMEN können wir Eingaben jedoch optisch erheblich anspruchsvoller gestalten, zum Beispiel so:

```
Dateiname  c:\gwbasic\adrerror.bas

Geben Sie einen Dateinamen ein und bestätigen
Sie mit RETURN oder brechen Sie mit ESC ab.
```

Und diese Abbildung ist noch recht bescheiden. Sie zeigt nicht, daß Rahmen und Kommentar (»Dateiname«) durch intensive Darstellung hervorgehoben werden und die Eingabezone invers ist. Dadurch wirkt das Ganze noch attraktiver. Probieren Sie's aus. Laden Sie QuickBASIC (und die User-Library). Kompilieren Sie CONVERT.BAS mit der Option »On Error« und wählen Sie den Menüpunkt »Laden« an. Laden Sie ein GW-BASIC-Programm, wählen Sie den Menüpunkt »Editieren« und anschließend den Menüpunkt »REM's ersetzen«. Dann erhalten Sie folgendes Bild:

```
      Datei  Editieren  Sonstiges
:
Flag=Ø
While Flag=Ø Or Instr(Ende$,A$)=Ø
  Locate Row,Col+Nr-1,1:
  Flag=Ø
Label14:   A$=Inkey$:If A$="" Then Goto Label14
  If Asc(Left$(A$,1))<32 Then Flag=1:A$=Right$(A$,1)
  Locate ,,Ø
  If Instr(Char$,A$)<>Ø   And Flag=Ø Then Gosub Label15
  If Instr(Sonder$,A$)<>Ø And Flag=1 Then Gosub Label16
Wend
:
:                Sind Sie sicher (j/n) ? j
Rem *** space
Rem funktion:
Rem              Beispiel: Aus <REM Kommentar> wird <' Kommentar>
Rem              Abbrechen(ESC) oder Bestätigen(RETURN)
:
While Right$(E$,1)=" " And Len(E$)<>Ø
  E$=Left$(E$,Len(E$)-1)
```

Hinter diesem »Standard-Eingabewindow« steckt ein nicht unbeträchtlicher Aufwand. Aber ich finde, er lohnt sich.

```
'       Standardeingabe
'       ---------------
'       Funktion: Übernimmt Eingabe in eigenem Window, das zeilenmittig
'                 zentriert wird
'       Hin     : Text$(..) : Kommentare, die unterhalb der Eingabe
'                               erscheinen
'                 Texte   : Anzahl zusätzlicher Kommentare
'                 Comment$  : Kommentar unmittelbar vor der Eingabe
'                 E$        : Vorgabe
'                 LenMax    : Maximale Eingabelänge
'                 Char$     : Eingabezeichen
'                 Back$     : Endezeichen
'       Zurück  : E$        : Eingabe
'                 Last$     : Endezeichen
'       Aufbau der Windows  :
'
'                             Kommentar  Eingabezone
'
'
'                             Zusatzkommentar (Text$(1))
'                             Zusatzkommentar (Text$(2))
'
```

Die Parameter <E$> (Vorgabe), <LenMax> (Länge der Eingabezone), <Char$> (zulässige Zeichen) und <Back$> (Endezeichen) kennen Sie von der Eingaberoutine.

Außerdem übergibt das aufrufende Programm <Text$(..)>, die »Zusatzkommentare« unterhalb der Eingabezone, <Texte>, die Kommentaranzahl und <Comment$>. Der Inhalt von <Comment$> wird unmittelbar vor der Eingabezone ausgegeben (im Beispiel »Dateiname«).

Wie zuvor WARTEN ermittelt auch STANDINPUT zuerst die Window-Parameter. Diesmal ist die Berechnung jedoch aufwendiger.

```
Sub StandInput(Texte, Comment$, E$, LenMax, Char$, Back$, Last$) Static


Sub StandInput(Texte, Comment$, E$, LenMax, Char$, Back$, Last$) Static

'       * Parameter des Gesamtwindows ermitteln *
  WLaenge = 6 + Texte
  WBreite = Len(Comment$) + 2 + LenMax + 1
  For i=1 to Texte
    If Len(Text$(i)) > WBreite Then WBreite = Len(Text$(i))
  Next i
  WBreite = WBreite + 4
  WCol = 40 - Int(WBreite / 2)
  WRow = 20 - Wlaenge
```

Die Window-Länge *6 + Texte* ergibt sich aus den Kommentaren plus den beiden Rahmenzeilen, den drei Zeilen des inneren Windows und einer Leerzeile zwischen dem inneren Window und den Kommentaren darunter.

Um die Window-Breite zu ermitteln, berechnet STANDINPUT zunächst die Breite der Eingabezeile (Länge des Kommentars + Leerzeichen + Rahmenzeichen (innerer Rahmen!) + Länge der Eingabezone + Rahmenzeichen (innerer Rahmen)). In dieser Berechnung ist der äußere Rahmen und je ein Zeichen Abstand zwischen Rahmen und Text noch nicht berücksichtigt!

In einer Schleife wird geprüft, ob einer (oder mehrere) der Kommentare länger ist als die komplette Eingabezeile. Nach der Schleife enthält <WBreite> die »Innenlänge« des Windows. Zu dieser Innenzeile wird 4 addiert, um auch die äußeren Rahmenzeichen und den Abstand zu berücksichtigen.

<WCol> und <WRow> werden wie zuvor in Abhängigkeit von der Window-Breite beziehungsweise Window-Länge berechnet. Das Standardeingabe-Window erscheint jedoch nicht wie bei WARTEN am unteren Bildschirmrand. Um für Abwechslung zu sorgen, ist es nach oben versetzt. Die untere Rahmenlinie befindet sich in Zeile 21 (*WRow = 20 – WLaenge*).

Praktisch unverändert gegenüber WARTEN ist das Retten des Untergrunds und die Rahmenausgabe.

```
'       * Untergrund retten/Rahmen ausgeben *
  Call PushScreen(WCol, WRow, WBreite, WLaenge)
  Call Rahmen(WCol, WRow, WBreite, WLaenge, Intensiv, 1)
```

Einen kleinen Unterschied gibt es doch: Der Rahmentyp 1 wird verwendet (gleiche Sorte wie bei den Pull-down-Menüs). Wenn Sie wollen, können Sie CONVERT.BAS jederzeit entsprechend ändern.

Die Ausgabe des Rahmeninhalts erfolgt in mehreren Schritten. Zuerst wird der innere Rahmen gezeichnet, der die Eingabezone umschließt.

```
'       * Parameter des Eingabewindows ermitteln *
  WInputCol     = WCol + 1 + Len(Comment$) + 2
  WInputRow     = WRow + 1
  WInputBreite = LenMax + 2
  WInputLaenge = 3

'       * Inhalt des Gesamtwindows ausgeben *
  Call Rahmen(WInputCol, WInputRow, WInputBreite, WInputLaenge, Normal, 3)
```

Die linke obere Ecke (<WInputCol> und <WInputRow>) wird in Abhängigkeit von der Position des äußeren Rahmens berechnet. Der benötigte »Offset« zu <WRow> und <WCol> wird ermittelt und addiert.

Die Breite ergibt sich aus der Länge der Eingabezone (*WInputBreite = LenMax + 2*) und die Länge ist immer drei (Eingabezeile plus zwei Rahmenzeilen).

RAHMEN gibt einen Rahmen vom Typ 3 in normaler Darstellung aus.

Übrigens: Das Retten des Untergrunds ist überflüssig. Der »Innenrahmen« befindet sich innerhalb des bereits geretteten Gebiets.

```
Call PrintF(WCol + 2, WRow + 2, Comment$, Intensiv)
For i=LBound(Text$) to LBound(Text$) + Texte - 1
  Call PrintF(WCol + 2, WRow + 4 + i, Text$(i), Normal)
Next i
```

Der Kommentar vor der Eingabezone wird ausgegeben. Nach der Ausgabe der Zusatzkommentare ist das Window vollständig, und die Eingaberoutine wird aufgerufen.

```
'       * Eingaberoutine aufrufen *
 Insert = False: PosAkt = 1
 Call LInput(E$, WInputCol + 1, WRow + 2, LenMax, Char$,_
             False, Back$, Insert, PosAkt, Last$, EditFlag)
```

Ich glaube nicht, daß ich die einzelnen Parameter ausführlich erläutern muß. Dafür wurde LINPUT zu intensiv besprochen. Ich nutze die Gelegenheit lieber, um Sie auf eine sehr ungewöhnliche Fehlerquelle aufmerksam zu machen:

Sie wundern sich vielleicht darüber, daß das Flag für den Darstellungsmodus (der sechste Parameter) direkt übergeben wird (<False>), das Flag <Insert> (Schreibmodus, achter Parameter) jedoch nicht. Nehmen wir an, wir würden folgenden Aufruf verwenden:

```
Call LInput(E$, WInputCol + 1, WRow + 2, LenMax, Char$,_
                False, Back$, False, PosAkt, Last$, EditFlag)
```

Auch bei diesem Aufruf wird als Schreibmodus <False> angegeben, jedoch ohne den Umweg über die zusätzliche Variable <Insert>. Wenn Sie sich an LINPUT erinnern, wissen Sie, daß diese Prozedur in der angegebenen Variablen den zuletzt eingestellten Schreibmodus übergibt.

Schaltet der Benutzer während der Eingabe in den Einfügemodus, gibt LINPUT in der angegebenen Variablen <False> den Wert <True> (=Einfügemodus eingeschaltet) zurück. Der Wert unserer »Booleschen Variablen« – die die Funktion einer Konstanten hat – wird verändert!

Hüten Sie sich davor, Variablen, die Sie als Konstanten verwenden (gleichbleibender Wert), per Referenz zu übergeben. Verwenden Sie eine »Zwischenvariable«, wenn die aufgerufene Prozedur die Variablen möglicherweise verändert.

Übrigens: Das Darstellungs-Flag ändert LINPUT niemals. Daher ist die Übergabe von <False> in diesem Fall unproblematisch.

Nach der Rückkehr aus LINPUT holt STANDINPUT den Untergrund zurück – die Eingabe ist beendet.

```
'       * Untergrund holen *
  Call PullScreen(WCol, WRow, WBreite, WLaenge)
End Sub
```

Teile des Bildschirms löschen (CLEARSCREEN)

Bevor wir die eigentliche Aufgabe von CONVERT besprechen, die Bearbeitung eines GW-BASIC-Programms, stelle ich Ihnen noch CLEARSCREEN vor. Diese Prozedur ist recht einfach, ist aber doch häufig sehr nützlich.

CLEARSCREEN löscht Bildschirmausschnitte. Übergeben wird eine »Start-« und eine »Endzeile«. CLEARSCREEN löscht diese beiden und alle dazwischenliegenden Zeilen.

Jetzt wissen Sie zwar, was CLEARSCREEN macht, aber noch nicht, wann diese Prozedur benötigt wird. In CONVERT soll die Menüleiste immer zu sehen sein. Üblicherweise löscht man nach Auswahl eines Menükommandos den Bildschirm, bevor der betreffende Programmteil ausgeführt wird. Mit CLEARSCREEN löschen wir einfach die Zeilen ab Nummer 2 und lassen die Menüleiste unangetastet.

```
'       ClearScreen
'       -----------
'       Funktion: Löscht angegebenen Zeilenbereich
'       Hin     : First : erste zu löschende Zeile
'                 Last  : letzte zu löschende Zeile

Sub ClearScreen(First, Last) Static
  For i=First to Last
    Call PrintF(1, i, Space$(8Ø), Normal)
  Next i
End Sub
```

Gelöscht wird eine Zeile durch Ausgabe von je 80 Leerzeichen. Die Ausgabe erfolgt mit unserer Assembler-Routine PRINTF, da es bei der gemütlichen Bildschirmausgabe von PCs doch eine größere Aktion ist, annähernd 2000 Zeichen (kompletter Bildschirm) auszugeben.

8.3 Verwendete Datenstrukturen

Wie bereits die Pull-down-Menüs ist auch CONVERT zu komplex, um einfach »draufloszuprogrammieren«. Überlegen wir uns zuerst, was es bedeutet, die in den Sprungbefehlen verwendeten Zeilennumern durch alphanumerische Label zu ersetzen. Als Denkmodell verwenden wir ein Demoprogramm,

das alle »kritischen« BASIC-Befehle enthält, Sprünge mit THEN, THEN GOTO und ON GOSUB.

```
100 INPUT "Wählen Sie ein Kommando (1-2)";K
101 IF K < 1 OR K > 2 THEN 100
102 ON K GOSUB 105, 120: GOTO 130
105 PRINT "K ist 1": RETURN
120 PRINT "K ist 2": RETURN
130 INPUT "Nochmal(j/n)";A$
140 IF A$ = "j" THEN GOTO 100
```

Für das Einlesen eines Programms verwendet CONVERT das Stringarray <Line$(..)>. In jedem Arrayelement wird genau eine Programmzeile gespeichert. Bereits beim Einlesen können die Zeilennummern entfernt werden:

```
Line$(1):  INPUT "Wählen Sie ein Kommando (1-2)";K
Line$(2):  IF K < 1 OR K > 2 THEN 100
Line$(3):  ON K GOSUB 105, 120: GOTO 130
Line$(4):  PRINT "K ist 1": RETURN
Line$(5):  PRINT "K ist 2": RETURN
Line$(6):  INPUT "Nochmal(j/n)";A$
Line$(7):  IF A$ = "j" THEN GOTO 100
```

Wir unterscheiden Programmzeilen ab jetzt nicht mehr durch ihre Zeilennummer, sondern anhand des zugehörigen Index im Array <Line$(..)>. Wenn ich in diesem Kapitel von »Zeile 100« spreche, meine ich nicht die Programmzeile 100, sondern die hunderste Zeile in <Line$(..)> (<Line$(100)>)!

Nach dem Entfernen der Zeilennummern wissen wir allerdings nicht mehr, wohin ein Sprung wie GOTO 102 führt. Daher speichern wir die Zeilennummern aller Programmzeilen vorläufig in einem eigenen Array, in <LineNr$(..)>. Jedes Element von <LineNr$(..)> enthält die Zeilennummer einer Programmzeile.

Zeilennummern		Programmzeilen	
LineNr$(1):	100	Line$(1):	INPUT "Kommando wählen (1-2)";K
LineNr$(2):	101	Line$(2):	IF K < 1 OR K > 2 THEN 100
LineNr$(3):	102	Line$(3):	ON K GOSUB 105, 120: GOTO 130
LineNr$(4):	105	Line$(4):	PRINT "K ist 1": RETURN
LineNr$(5):	120	Line$(5):	PRINT "K ist 2": RETURN
LineNr$(6):	130	Line$(6):	INPUT "Nochmal(j/n)";A$
LineNr$(7):	140	Line$(7):	IF A$ = "j" THEN GOTO 100

Dieser Arrayzustand ergibt sich nach dem Einlesen. Die Zeilennummern sind entfernt. Im nächsten Schritt wird das Array analysiert. Um im folgenden Begriffsverwirrungen zu vermeiden:

- »Sprünge« sind Markierungen, die einem Sprungbefehl folgen (*GOTO 1234* oder *GOTO Label1*).
- »Zielzeilen« sind Programmzeilen, die das Ziel eines Sprungbefehls sind (*130 INPUT "Nochmal(j/n)";A$*).
- Ein »Label« ist die Markierung einer Zielzeile, zum Beispiel die zugehörige Zeilennummer (die später durch ein alphanumerisches Label ersetzt wird).

Bei der Analyse werden mehrere Tabellen angelegt:

1. Eine Tabelle <L$..)>, die alle vorkommenden Label enthält, also die Zeilennummern aller angesprungenen Programmzeilen.
2. Eine weitere Tabelle <LLine(..)>, die die Indizes dieser Zielzeilen enthält.
3. Eine dritte Tabelle <LZahl(..)>, die angibt, wie oft im gesamten Programm ein Sprung zu einem bestimmten Label vorkommt.
4. Eine zweidimensionale Tabelle <LTab(.., ..)>, die alle Sprünge zu jedem Label enthält. Das heißt, diese Tabelle enthält für jedes Label eine Liste mit den Indizes (Index in <Line$(..)>) aller Zeilen, in denen ein Sprung zu diesem Label vorkommt.

Auf unser Demoprogramm angewandt ergeben sich folgende Tabelleneinträge:

Label	Zielzeile	Sprunganzahl	Zeilen mit Sprüngen	
L$(1)=100	LLine(1)=1	LZahl(1)=2	LTab(1,1)=2	LTab(1,2)=7
L$(2)=105	LLine(2)=4	LZahl(2)=1	LTab(2,1)=3	
L$(3)=120	LLine(3)=5	LZahl(3)=1	LTab(3,1)=3	
L$(4)=130	LLine(4)=6	LZahl(4)=1	LTab(4,1)=3	

Wenn Sie sich noch einmal das Demoprogramm anschauen, fällt es Ihnen leichter, meinen Erläuterungen zu folgen.

Zur Interpretation: Das Programm enthält vier verschiedene Label (Markierungen von Zielzeilen), <L$(1)> bis <L$(4)>.

Das Array <LLine(..)> enthält den Index jeder Zielzeile im Array <Line$(..)>. LLine(1)=1 bedeutet, daß das erste Label »100« die Zeile Nummer 1 markiert (<Line$(1)>).

<LZahl(..)> gibt an, wie viele Sprünge es zu jedem der vier Label gibt. Zum ersten Label (»100«) existieren zwei Sprünge (LZahl(1)=2).

Diese beiden Sprünge befinden sich in den Zeilen mit den Indizes 2 (LTab(1,1)=2) und 7 (LTab(1,2)=7), also in <Line$(2)> und <Line$(7)>.

8.4 Programmentwicklung

Es geht los. Sie kennen die verwendeten Datenstrukturen; beginnen wir mit der Umsetzung.

8.4.1 Globale Variablen in komplexen Programmsystemen

Der Inhalt der Tabellen wird von verschiedenen Prozeduren verwendet. Da es ziemlich unsinnig ist, all diese Arrays beim Aufruf immer wieder zu übergeben, deklariere ich die Arrays am Programmanfang als globale Variablen.

```
'       ************************************************
'       * Convert: GW-BASIC nach QuickBASIC konvertieren *
'       ************************************************

'       (C) Said Baloui, 1987


REM $INCLUDE: 'comdef.bas'

Common Shared Line$(1)          'Programmzeilen (dynamisches Array)
Common Shared L$(1), LLine(1)   'Label und Nummer der zugehörigen Zeile
Common Shared LZahl(1)          'Anzahl der Sprünge zu diesem Label
Common Shared LTab(2)           'Zeilen, in denen sich die Sprünge zu
                                 einem bestimmten Label befinden
Common Shared Text$(1)          'Verschiedene Texte, z.B. Fehlermeldungen
Common Shared LGesamt           'Gesamtanzahl der Label
Common Shared LineZahl          'Anzahl der Programmzeilen
Common Shared File$, Dir$       'Dateiname und aktuelles Verzeichnis
Common Shared LineMax, JumpMax  'Max. Anzahl an Sprüngen

LineMax = 1000: LMax = 100: JumpMax = 50  'Ändern: Verkleinern, wenn Ihr
Dim L$(LMax), LLine(LMax), LZahl(LMax)    'RAM zu klein ist, vergrößern,
Dim LTab(LMax, JumpMax)                   'wenn die Arrays für Ihre GW-
Dim Text$(10)                             'BASIC-Programme zu klein sind
File$ = ""

REM $INCLUDE: 'init.bas'
```

Dieser Programmabschnitt ist ungeheuer wichtig. Er zeigt, wie globale Variablen innerhalb eines(!) Programms in Verbindung mit für alle(!) Programme globalen Variablen deklariert und definiert werden.

Erinnern Sie sich: COMMON SHARED-Anweisungen müssen sich vor allen ausführbaren Anweisungen befinden!

Das heißt, auf keinen Fall darf dieser Programmkopf so gestaltet sein:

```
REM $INCLUDE: 'comdef.bas'
REM $INCLUDE: 'init.bas'
Common Shared Line$(1)          'Programmzeilen (dynamisches Array)
Common Shared L$(1), LLine(1)   'Label und Nummer der zugehörigen Zeile
...
...
```

QuickBASIC liefert beim Kompilieren eine Fehlermeldung! Die Ursache: REM $INCLUDE: 'init.bas' bindet die Initialisierungs-Datei ein, in der den globalen Variablen von COMDEF.BAS Werte zugewiesen werden.

Zuweisungen sind ausführbare Anweisungen. Und diesen ausführbaren Anweisungen folgen nun COMMON SHARED-Anweisungen!

Verwenden Sie daher bitte immer folgende Struktur:

```
-------------------------------------------------------------
$INCLUDE: 'Dateiname'       'Einbindung der für alle (!)
                            'Programme globalen Variablen
-------------------------------------------------------------
COMMON SHARED Variable      'Deklaration der für dieses (!)
COMMON SHARED Variable      'Programm globalen Variablen
...
...
COMMON SHARED Variable
-------------------------------------------------------------
Variable = Ausdruck         'Wertzuweisungen an die globalen
Variable = Ausdruck         'Variablen dieses Programms
...                         '(natürlich nur nötig, wenn diese
...                         'Variablen tatsächlich initialisiert
Variable = Ausdruck         'werden sollen)
-------------------------------------------------------------
REM $INCLUDE: 'init.bas'    'Wenn nötig, Einbindung der Initia-
                            'lisierungsdatei für die generell
                            'globalen Variablen
-------------------------------------------------------------
```

Auf diese Weise können Sie Variablen für ein bestimmtes Programm global definieren und zugleich Dateien einbinden, in denen wiederum globale Variablen (für mehrere oder alle Programme) deklariert werden.

Das Prinzip ist ganz einfach: Erst deklarieren (COMMON SHARED) und dann initialisieren (Zuweisungen oder Dimensionierungen).

Außer den bereits besprochenen Arrays werden noch einige weitere globale Variablen deklariert und initialisiert:

- <Text$(..)>: Array, das Meldungen und Kommentare enthält (WARTEN und STANDINPUT).
- <LGesamt>: Gesamtanzahl der im GW-BASIC-Programm enthaltenen Label (<L$(1)> bis <L$(LGesamt)>).

- <LineZahl>: Anzahl der Programmzeilen (<Line$(1)> bis <Line$(LineZahl)>).
- <File$>: Vom Benutzer eingegebener Dateiname.
- <Dir$>: Aktuelles Verzeichnis.
- <LineMax>: Maximalzahl zu verarbeitender Programmzeilen.
- <JumpMax>: Maximalzahl zu verarbeitender Sprünge.

Einige Variablen müssen Sie eventuell ändern und Ihnen größere beziehungsweise kleinere Werte zuweisen. <LineMax> bestimmt die Größe des Arrays <Line$(..)>, in dem das zu konvertierende GW-BASIC-Programm gespeichert wird. Der verwendete Wert 1000 sollte für nahezu alle Programme ausreichen. Sollten Sie dennoch einmal ein Programm mit mehr als 1000 Zeilen konvertieren, vergrößern Sie einfach <LineMax>.

<LMax> ist die maximal verarbeitbare Labelanzahl und <JumpMax> gibt an, wie viele Sprünge zu einem Label das Programm höchstens verarbeitet. Vergrößern Sie bitte auch diese Werte, wenn sich ein Programm nicht konvertieren läßt.

Eventuell besteht die Notwendigkeit, die Arrays nicht zu vergrößern, sondern zu verkleinern. Mein eigener Rechner besitzt nach Abzug aller residenten Utilities eine freie Hauptspeicherkapazität von rund 350 Kbyte.

Dieser Platz reicht für QuickBASIC und die residente User-Library, für den Sourcetext von CONVERT, das erzeugte Kompilat und für ein immerhin 25 Kbyte großes Testprogramm, das ich konvertierte.

Sollte die freie Speicherkapazität Ihres Rechners deutlich geringer sein, bekommen Sie Schwierigkeiten mit CONVERT. Die letzte Notlösung ist dann eine Verkleinerung der Arrays. Am wirkungsvollsten ist diese Maßnahme beim »speicherfressenden« zweidimensionalen Array <LTab(LMax, JumpMax)>. Verringern Sie also notfalls die Werte von <LMax> und <JumpMax>.

Aber bevor Sie diesen Schritt unternehmen, probieren Sie erst aus, wieviel Platz Sie gewinnen, wenn Sie alle speicherresidenten Programme entfernen – Sidekick, Norton Commander, Lightning und was man im »Programmierer-Alltag« sonst noch braucht.

Übrigens: Vermissen Sie die Dimensionierung des Arrays <Line$(..)>? Sie wissen inzwischen, daß dieses Array von zentraler Bedeutung ist, da es die einzelnen Programmzeilen des eingelesenen Programms aufnimmt.

Da wir sowieso schon Speicherplatzprobleme haben, wollte ich vermeiden, <Line$(..)> zum Beispiel mit dem Wert 1000 zu dimensionieren und damit

unnötig Platz zu verschwenden. Die wenigsten Programme besitzen 1000 oder mehr Programmzeilen und das Array würde erheblich mehr Platz als nötig einnehmen. <Line$(..)> wird dynamisch dimensioniert. Zuerst wird geprüft, wie viele Programmzeilen das angegebene GW-BASIC-Programm besitzt. <Line$(..)> wird anschließend genau in der benötigten Größe angelegt, ohne jede Platzverschwendung!

Das Hauptmenü

Das Auswahlmenü beruht auf den Pull-down-Menüs. Drei Einzelmenüs werden definiert und dann PULLDOWN aufgerufen.

```
'       Hauptprogramm
'       -------------
On Error Goto ErrorHandling

Header$   = "      Datei  Editieren  Sonstiges
"
Menue$(1) = "Laden/Speichern/"
Menue$(2) = "Liste/Editieren/REM ersetzen/Gross/Klein/"
Menue$(3) = "Directory/Zugriffspfad/-/Programm verlassen/"

MainLoop:Call PullDown(Menue, Befehl)
If Menue = 1 Then
  If Befehl = 1 Then Call LoadFile
  If Befehl = 2 And LineZahl <> Ø Then Call SaveFile
ElseIf Menue = 2 And LineZahl <> Ø Then
  If Befehl = 1 Then Call LListe
  If Befehl = 2 Then Call LEdit
  If Befehl = 3 Then Call ReplaceRem
  If Befehl = 4 Then Call GrossKlein
ElseIf Menue = 3 Then
  If Befehl = 1 Then Call Directory
  If Befehl = 2 Then Call EditDir
  If Befehl = 3 Then Call EndProg
End If
Goto MainLoop
```

Nach Rückkehr aus PULLDOWN wird anhand des zuletzt aktiven Menüs und des selektierten Kommandos zu einer von neun Prozeduren verzweigt. Die aufgerufene Prozedur führt das betreffende Kommando aus. Anschließend werden erneut die Pull-down-Menüs aufgerufen (Goto MainLoop).

8.4.2 Fehlerbehandlung

Am Anfang des Hauptprogramms befindet sich die Anweisung »On Error Goto ErrorHandling«. CONVERT arbeitet mit eingeschalteter Fehlerabfrage. Wenn ein Fehler auftritt, wird zur Prozedur ERRORHANDLING am

Programmende verzweigt. Wegen der Fehlerbehandlung muß bei der Kompilierung von CONVERT die Option »On Error« aktiviert sein!

```
'       ErrorHandling
'       -------------
'       Funktion: Behandelt das Auftreten von Fehlern
'                 Fehler wird gemeldet und zum Menü verzweigt

ErrorHandling:
Text$(1) = "Fehler Nummer" + Str$(Err) + " aufgetreten"
If Err = 76 Then Text$(1) = "Dieses Verzeichnis existiert nicht"
If Err = 53 Then Text$(1) = "Diese Datei existiert im nicht im aktuellen"_
   "Verzeichnis"
Call Warten(1, Key$, Invers)
Resume MainLoop
```

ERRORHANDLING behandelt Fehler auf recht primitive Art und Weise. <Text$(1)> wird ein entsprechender Hinweis mit der Fehlernummer zugewiesen und dann die Prozedur zur Ausgabe von Meldungen aufgerufen, WARTEN. Die Meldung wird invers ausgegeben, und – wie der Name sagt – WARTEN wartet auf eine Taste. Danach wird mit RESUME zur Hauptschleife zurückgekehrt, zur Ausgabe der Pull-down-Menüs.

Nur in zwei Fällen erhalten Sie eine ausführlichere Fehlermeldung: Wenn der Fehler Nummer 76 (»Verzeichnis nicht gefunden«) oder 53 (»Datei nicht gefunden«) auftritt.

Achtung: Beim Kompilieren von CONVERT muß die Option »On Error« aktiv sein (vergißt man leicht, wie ich bestätigen kann)!

Ich empfehle Ihnen, in Ihren eigenen Programmen die Fehlerbehandlung zu optimieren, ausführlichere Meldungen auszugeben oder gar mit *RESUME NEXT / RESUME Label* gezielt je nach Fehlerart bestimmte Prozeduren aufzurufen.

Aber vergessen sie bitte nicht, bei Verwendung von *RESUME NEXT* vor dem Kompilieren die zugehörige Kompilieroption zu aktivieren!

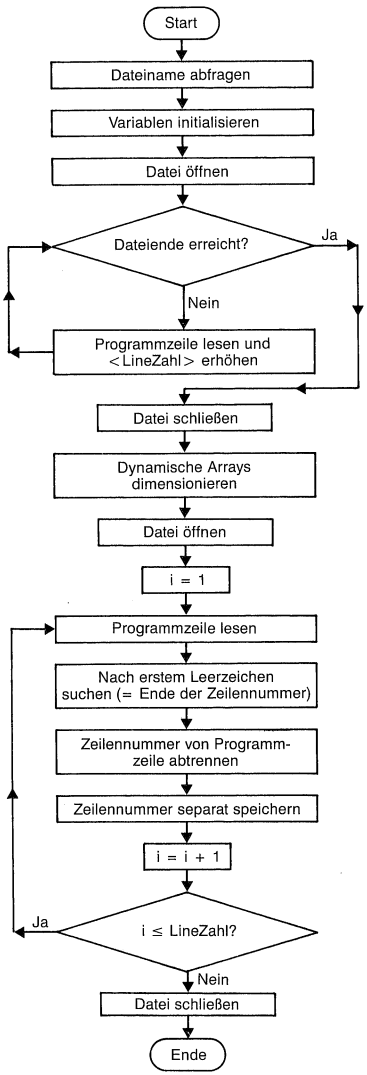


Bild 19 *Laden eines GW-BASIC-Programms*

In den hinter uns liegenden Projekten stellte ich immer zuerst die einfacheren Prozeduren vor und erst am Schluß die komplexen. Ich wollte Ihnen damit die Gelegenheit geben, mit dem Programm »warm zu werden«, bevor wir die problematischen Teile behandeln.

Der Aufbau von CONVERT verhindert diese Vorgehensweise. Beim Laden wird das GW-BASIC-Programms analysiert, nach Sprüngen durchsucht und Zeilennummern durch Label ersetzt. Und da alle folgenden Prozeduren auf dieser Umwandlung des Originalprogramms aufbauen, müssen wir uns zuerst um die Prozedur LOADFILE kümmern.

```
'       LoadFile
'       --------
'       Funktion: Lädt Programmdatei, entfernt die Zeilennummern und
'                 ersetzt Sprungziele und Sprünge durch 'Standard-Label'
'                 (Label1, Label2 etc.)


Sub LoadFile Static

'       * Dateiname ? *
  Locate 3, 1: Files
  Text$(1) = "Abbrechen (ESC) oder Bestätigen (RETURN)"
  Call StandInput(1, "Dateiname", File$, 4Ø, Alpha$, zEsc$ + zReturn$,_
  Last$)
  If Last$ = zEsc$ Then Exit Sub
```

Der Benutzer wird nach dem Namen des zu konvertierenden Programms gefragt, natürlich unter Verwendung von STANDINPUT. Um Ihnen die Auswahl zu erleichtern, gibt *Locate 3, 1: Files* das Directory ab der dritten Bildschirmzeile aus.

Als Endezeichen werden <zEsc$> und <zReturn$> definiert. Mit RETURN beenden Sie eine Eingabe »normal«, mit ESC brechen Sie die aktuelle Funktion ab und kehren zum Menü zurück (*If Last$ = zEsc$ Then Exit Sub*).

Nach Rückkehr aus STANDINPUT enthält die globale Variable <File$> den Dateinamen. Vor dem Laden des Programms werden noch kleinere Vorbereitungen getroffen.

```
'       * Vorbereitungen zum Einlesen *
  For i=1 to JumpMax
    LZahl(i) = Ø
  Next i
  LineZahl = Ø
```

<LineZahl> – die Anzahl der Programmzeilen – und alle Elemente von <LZahl(..)> (Anzahl der Sprünge zu jedem der vorkommenden Label) werden mit 0 initialisiert. Sie können nun argumentieren, daß diese Variablen wie alle numerischen Variablen nach dem Starten von CONVERT sowieso den Ausgangswert 0 besitzen. Stimmt; dabei ist jedoch nicht berücksichtigt, daß Sie mit CONVERT mehrere Programme nacheinander laden und »bearbeiten« können. Und nach dem ersten Ladevorgang besitzen

<LineZahl> und die Elemente von <LZahl(..)> nicht mehr den Ausgangswert 0!

Theoretisch könnten wir die Datei nun laden. Aber erinnern Sie sich: <Line$(..)> wird – um keinen Platz zu verschwenden – nur so groß wie nötig angelegt. Also ermitteln wir zuerst, wie viele Programmzeilen die angegebene Datei enthält.

```
'      * Anzahl Programmzeilen ermitteln/dynamische Arrays dimensionieren *
  F$ = File$: If Instr(F$, ".") = Ø Then F$ = F$ + ".bas"
  Open F$ For Input As #1
  While Not Eof(1)
    Line Input #1, A$
    LineZahl = LineZahl + 1
  Wend
  Close #1
```

<File$> wird in <F$> kopiert und die Datei geöffnet. Zuvor wird überpüft, ob <F$> einen Punkt enthält. Wenn ja, darf man davon ausgehen, daß der Benutzer den kompletten Dateinnamen angab, inklusive der Erweiterung ".bas". Enthält <F$> keinen Punkt, hängt das Programm diese Erweiterung einfach an.

Solange das Ende der Datei nicht erreicht ist, liest die Routine immer wieder eine Programmzeile in die Variable <A$> ein und erhöht <LineZahl> um eins.

8.4.3 Redimensionieren dynamischer Arrays

Nach dem Verlassen der Schleife kennen wir die genaue Anzahl der Programmzeilen und können die dynamischen Arrays dimensionieren.

```
REM $DYNAMIC
Redim Line$(LineZahl), LineNr$(LineZahl)
```

Der Metabefehl $DYNAMIC bewirkt, daß alle mit einem nachfolgenden DIM- oder REDIM-Befehl dimensionierten Arrays dynamisch sind. Dynamische Arrays können Sie jederzeit löschen und wieder neu dimensionieren. Das Löschen mit ERASE ist zwar auch bei statischen Arrays möglich, hat jedoch eine völlig andere Bedeutung: Die Variablenwerte werden gelöscht (null beziehungsweise Leerstring).

Das Löschen dynamischer Arrays bewirkt eine Freigabe des belegten Speichers. Das Array wird vollständig entfernt!

Außer <Line$(..)> wird ein weiteres Array in der ermittelten Größe angelegt. In <LineNr$(..)> speichern wir die Zeilennummer jeder einzelnen Programmzeile. Dieses Array benötigen wir nur vorübergehend. Nach beendetem Einlesen werden wir es mit Erase LineNr$ entfernen.

Sicher fragen Sie sich, warum die beiden Arrays nicht mit DIM, sondern mit REDIM angelegt werden. REDIM ist eine Kombination aus ERASE und DIM. Das Array wird also zuerst gelöscht, falls es bereits existiert.

Da <Line$(..)> im Gegensatz zu <LineNr$(..)> nicht nur vorübergehend benötigt wird, werden wir dieses Array nach dem Einlesen nicht mit ERASE löschen. Einmal angelegt, bleibt es während des gesamten Programmlaufs erhalten.

Nehmen wir nun an, Sie wollen nach der Behandlung eines GW-BASIC-Programms ein weiteres konvertieren. Dann wird die Prozedur LOADFILE erneut durchlaufen. Wenn wir statt REDIM die DIM-Anweisung verwenden, erhalten wir eine Fehlermeldung. Denn QuickBASIC kann ein bereits existierendes Array natürlich nicht ein zweites Mal anlegen!

```
'     * Prog-Zeilen lesen, dabei Zeilennummern entfernen *
  Open F$ For Input As #1
  For i=1 to LineZahl
    Line Input #1,Line$(i)
    j = Instr(Line$(i), " ")
    LineNr$(i) = Left$(Line$(i), j - 1)
    Line$(i)   = Mid$(Line$(i), j + 1) + " "
  Next i
  Close #1
```

Nachdem die beiden Arrays dimensioniert sind, können wir das Programm endlich laden.

Wieder wird eine Schleife verwendet, diesmal jedoch eine FOR..NEXT-Schleife, da wir die Anzahl der einzulesenden Zeilen inzwischen kennen (<LineZahl> Programmzeilen).

Bereits beim Einlesen entfernt CONVERT die Zeilennummern. Mit INSTR wird das erste Leerzeichen in der aktuellen Zeile <Line$(i)> gesucht. Es befindet sich immer direkt nach der Zeilennummer. Der linke Teil von <Line$(i)> bis zu dieser Position wird in <LineNr$(i)> gespeichert, also die Zeilennummer.

<Line$(i)> wird der rechte Teil der Programmzeile ab dem Leerzeichen nach der Zeilennummer zugewiesen. Was nicht jeder weiß: MID$ ohne Angabe der Länge übergibt den kompletten String ab der angegebenen Startposition. Die Anweisung

```
Mid$(Line$(i), j + 1)
```

ist daher identisch mit

```
Right$(Line$(i), j + 1, Len(Line$(i) - j)
```

aber erheblich einfacher und kürzer. Warum an jede Programmzeile ein Leerzeichen angehängt wird, erkläre ich Ihnen später. Diese Kleinigkeit wird uns die Arbeit sehr vereinfachen. Merken Sie sich bitte vorläufig nur, daß jede Zeile in <Line$(..)> mit einem Leerzeichen endet.

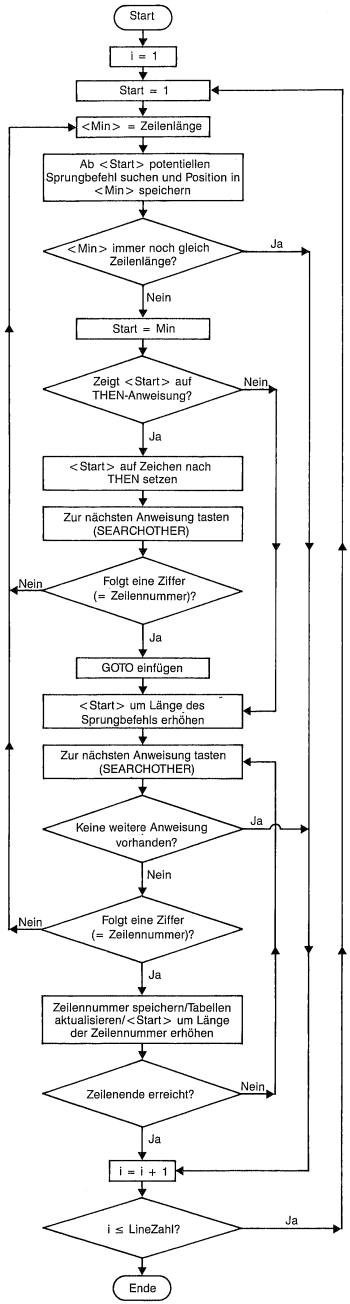


Bild 20 *Programm analysieren*

Auf den folgenden Programmteil bin ich nicht gerade stolz. Es ist eine riesengroße Schleife, in der eine Vielzahl von Aktionen stattfinden. Die Schleife enthält einfach zuviel, um die Übersicht zu wahren und widerspricht der von mir geschätzten modularen Programmierweise. Sie enthält sogar mehrere der »verpönten« GOTO-Anweisungen.

Aber ich muß gestehen, daß ich einfach keine andere Möglichkeit sah (vielleicht gelingt es Ihnen, mehr »Form« in diesen Abschnitt zu bringen). Also versuchen wir einfach, dieses »Dickicht« so gut wie möglich zu durchdringen.

Die Schleife analysiert der Reihe nach alle in <Line$(..)> eingelesenen Programmzeilen. <Ziffer$> wird uns helfen, Zeilennummern nach Sprunganweisungen zu entdecken. In <LGesamt> speichern wir die Gesamtanzahl aller entdeckten Label (=Zeilennummern, die angesprungen werden).

```
'        *** Label ermitteln ***
  Ziffer$ = "1234567890"
  LGesamt = 0
  For i=1 to LineZahl
    Start = 1
```

<Start> ist die wichtigste Variable in diesem Programmteil. Sie ist ein Zeiger auf die aktuelle Position innerhalb der Zeile. Und da wir mit der Analyse natürlich ab dem ersten Zeichen beginnen, setzen wir <Start> auf den Ausgangswert 1. Ab <Start> suchen wir nun nach einem Sprungbefehl.

```
'        * Sprungbefehl suchen *
SearchJump:
    P1 = Instr(Start, Line$(i), "GOTO")
    P2 = Instr(Start, Line$(i), "GOSUB")
    P3 = Instr(Start, Line$(i), "RESUME")
    P4 = Instr(Start, Line$(i), "THEN")
    Min = Len(Line$(i))
    If P1 <> 0 Then Min = P1: L = 4
    If P2 <> 0 And P2 < Min Then Min = P2: L = 5
    If P3 <> 0 And P3 < Min Then Min = P3: L = 6
    If P4 <> 0 And P4 < Min Then Min = P4: L = 4
    If Min = Len(Line$(i)) Then Goto NextLine Else Start = Min
```

CONVERT sucht nach allen vier möglichen Sprungbefehlen (GOTO/GOSUB/RESUME/THEN). Auch THEN ist ein eigener Sprungbefehl, da nicht zwangsläufig ein GOTO nach THEN folgen muß: *THEN GOTO 100* oder *THEN 100* (und diese Kurzschreibweise wird uns noch viel Ärger bereiten).

<P1> bis <P4> enthalten nun die Positionen innerhalb <Line$(i)>, an denen die vier Sprungbefehle zum erstenmal ab der Position <Start> auftreten (oder aber 0, wenn einer der Sprungbefehle nicht vorkommt).

<Min> wird mit der Länge der Programmzeile initialisiert. Anschließend wird <Min> der kleinste Wert von <P1> bis <P4> zugewiesen, der

ungleich 0(!) ist. Wenn überhaupt ab <Start> ein Sprungbefehl vorkommt, enthält <Min> nun die Position des ersten Sprungbefehls (es können ja mehrere sein!).

Enthält <Min> dagegen immer noch den Ausgangswert *Len(Line$(i))*, existiert ab Position <Start> kein Sprungbefehl in der betreffenden Zeile. Wir übergehen die Zeile und analysieren die nächste (Sprung zu NEXTLINE).

Ansonsten wird unser Zeiger <Start> auf die Position des ersten Sprungbefehls gesetzt (*If Min = Len(Line$(i)) Then Goto NextLine Else Start = Min*).

Beachten Sie, daß die Variable <L> nun einen Wert enthält, der genau der Länge des gefundenen Sprungbefehls entspricht! In Kürze werden wir <L> zu <Start> addieren. <Start> weist anschließend auf das erste Zeichen nach dem Sprungbefehl.

Wenn <Start> mit <P4> identisch ist, entdecken wir eine THEN-Anweisung. Und die macht uns das Leben sehr schwer! THEN heißt nicht von vornherein, daß hier ein Sprung erfolgt (*THEN X=X+1*). Wir müssen erst prüfen, ob eine Zeilennummer folgt. Wenn tatsächlich eine Zeilennummer folgt, liegt zwar ein Sprung vor, jedoch dürfen wir die betreffende Zeilennummer nicht einfach durch ein alphanumerisches Label ersetzen!

QuickBASIC gestattet die Verwendung von Label nach GOTO, GOSUB und RESUME, jedoch nicht nach THEN. Die Anweisung *THEN Label1* ist falsch! Im Gegensatz zu *THEN GOTO Label1*. Kommt nach THEN eine Zeilennummer, müssen wir daher hinter THEN eine GOTO-Anweisung »einflicken«.

Die Behandlung von THEN wird noch verwirrender, wenn wir berücksichtigen, daß das Wort THEN womöglich nur Teil einer Sprunganweisung *THEN GOTO*, *THEN GOSUB* oder *THEN RESUME* ist!

In diesem Fall müssen wir THEN einfach übergehen und uns zur eigentlichen Sprunganweisung vortasten.

Merken Sie sich die möglichen Fälle:

1. THEN ohne nachfolgenden Sprung
2. THEN als Teil einer THEN GOTO-, THEN GOSUB- oder THEN RESUME-Anweisung
3. THEN mit folgendem Sprung (THEN 100)

```
'       * Sonderbehandlung von THEN *
    If Start = P4 Then
      Start = Start + 4
      Call SearchOther(Line$(i), Terminator$, Start)     'Zeilennummer?
      If Instr(Ziffer$, Mid$(Line$(i), Start, 1)) <> Ø Then_
        Line$(i) = Left$(Line$(i), Start - 1) + "GOTO " +_
                   Mid$(Line$(i), Start)_
        Else Goto SearchJump
    End If
    Start = Start + L
```

Die Routine zur »Spezialbehandlung« von THEN prüft zuerst, ob eine THEN-Anweisung vorliegt. Das ist der Fall, wenn <Start> – unser aktueller Positionszeiger – mit <P4> identisch ist (*If P4 <> 0 And P4 < Min Then Min = P4: L = 4*). Wenn ja, wird <Start> erhöht, um die Länge von THEN. <Start> zeigt nun auf das erste Zeichen nach THEN.

SEARCHOTHER sucht das erste Nicht-Trennzeichen (das nicht in <Terminator$> enthalten ist), also das erste Zeichen der Anweisung nach THEN.

Nun prüft CONVERT, ob THEN eine Zeilennummer folgt. Die Zeilennummer erkennen wir daran, daß das aktuelle Zeichen Nummer <Start> eine Ziffer ist und *Instr(Ziffer$, Mid$(Line$(i), Start, 1))* einen Wert ungleich null übergibt.

Ist das der Fall, müssen wir zwischen THEN und der Zeilennummer das Wort GOTO einfügen. <Line$(i)> wird dem linken Teil bis zur aktuellen Position <Start>, der Anweisung GOTO und dem rechten Teil ab <Start> gebildet.

Wenn ein THEN entdeckt wurde, aber keine Zeilennummer folgt, wird die Anweisung *Else Goto SearchJump* ausgeführt und auf diese Weise werden die Fälle 1. und 2. elegant behandelt.

Folgendes passiert: <Start> zeigt auf das Zeichen nach THEN. Ab dieser Position wird nach einem Sprung gesucht. Wenn ein Sprung folgt, wird <Start> auf das Ende nach Sprung zeigen. Damit ist Fall 2. behandelt. <Start> zeigt auf die Position der GOTO, GOSUB oder RESUME-Anweisung, die dem Wort THEN folgt.

Gibt es keine solche Anweisung (*Start=Len(Line$(i))*), wird zu NEXTLINE verzweigt. Da THEN kein Sprung folgt, ist die aktuelle Zeile behandelt.

Nach diesem Programmteil – genauer: nach End If – zeigt <Start> auf das erste Zeichen des Sprungbefehls und <L> enthält die Länge des »Sprungwortes« (4, 5 oder 6).

Der Offset <L> wird zu <Start> addiert. <Start> zeigt auf das erste Zeichen nach dem Sprungbefehl, also auf ein Leerzeichen. CONVERT ermittelt nun die Zeilennummer.

```
'       * Zeilennummer ermitteln *
SearchNr:
    Call SearchOther(Line$(i), Terminator$, Start) 'Zeilennummer?
    If Start = Ø Then Goto NextLine
    If Instr(Ziffer$, Mid$(Line$(i), Start, 1)) = Ø Then Goto SearchJump
```

Zwischen einem Sprungbefehl und der Zeilennummer können sich beliebig viele Leerzeichen befinden (*GOTO 1234*). Um sich zur Zeilennummer vorzutasten, muß CONVERT alle Leerzeichen überlesen. Die einfachste Methode: SEARCHOTHER sucht das erste Nicht-Trennzeichen ab <Start> und weist die betreffende Position wieder <Start> zu.

Bitte »übersehen« Sie vorläufig die folgenden Anweisungen. Merken Sie sich nur, daß <Start> in diesem Programmabschnitt auf das erste Zeichen einer Zeilennummer nach einem Sprungbefehl zeigt.

Diese Zeilennummer, dieses »Label«, werden wir nun in der Labeltabelle <L$(..)> speichern. <LGesamt> – die Anzahl aller Label – wird aktualisiert. In <LTab(.., ..)> speichern wir die Nummer der Programmzeile und in <LZahl(..)>, wie oft das entdeckte Label bisher im Programmtext vorkommt.

```
'       * Zeilennummer in Labeltabelle speichern *
    Ende = Start
    Call SearchOther(Line$(i), Ziffer$, Ende)
    L$ = Mid$(Line$(i), Start, Ende - Start)
    If L$ = "Ø" And Left$(Line$(1), 1) <> "Ø" Then Goto Jump
    j = 1
    While L$(j) <> L$ And j <= LGesamt
      j = j + 1
    Wend
    LZahl(j) = LZahl(j) + 1
    LTab(j, LZahl(j)) = i
    If j > LGesamt Then LGesamt = j: L$(LGesamt) = L$
```

Wir kennen die Startposition der Zeilennummer. <Start> weist auf das erste Zeichen. <Ende> soll angeben, wo die Zeilennnummer endet und wird mit <Start> initialisiert.

SEARCHOTHER sucht ab <Ende> nach dem Auftreten eines nicht in <Ziffer$> enthaltenen Zeichens, zum Beispiel einem Leerzeichen (letztes Zeichen der Programmzeile!) oder einem Doppelpunkt (*GOTO 1234* beziehungsweise *GOTO 1234:REM test*).

<L$> wird die Zeilennummer zugewiesen, der mittlere Teil von <Line$(..)> ab <Start> in der Länge Ende – Start.

Nun folgt die recht komplexe Aktualisierung der verschiedenen Tabellen. Zuerst müssen wir jedoch einen Spezialfall berücksichtigen: die Fehlerbehandlung mit *ON ERROR GOTO 0*!

ON ERROR GOTO 0 ist eine spezielle Art der Fehlerbehandlung und bedeutet keineswegs, daß nun zur Zeile 0 verzweigt wird. Ganz im Gegenteil: wahrscheinlich existiert gar keine Zeile 0!

Dieses »Speziallabel« werden wir daher einfach nicht berücksichtigen. Wir prüfen, ob die aktuelle Zeilennummer die Nummer 0 ist. Wenn ja, ist die gesuchte Zielzeile mit der Zeilennummer 0 zwangsläufig die erste Programmzeile. Enthält die erste Programmzeile jedoch eine andere Zeilennummer, liegt die Anweisung *ON ERROR GOTO 0* vor und Zeile 0 existiert überhaupt nicht. In diesem Fall überspringen wir die Behandlung dieses Labels.

```
If L$ = "Ø" And Left$(Line$(1), 1) <> "Ø" Then Goto Jump
```

Kommen wir zum »Normalfall«. Zuerst müssen wir herausbekommen, ob die Zeilennummer <L$> einen Sprung zu einem noch nicht in den Tabellen vermerkten Label kennzeichnet.

Die Tabelle <L$(..)> enthält alle bisher entdeckten Label. In <L$(..)> nehmen wir nur neue Label auf. Daher müssen wir prüfen, ob <L$> ein neues Label oder aber bereits in dieser Tabelle enthalten ist. Wenn die WHILE-Schleife verlassen wird, enthält der Schleifenzähler <j> entweder die Nummer des – bereits vorhandenen – Labels in <L$(..)> oder ist um eins größer als <LGesamt>, die Gesamtanzahl aller in <L$(..)> gespeicherter Label.

<LZahl(j)> gibt an, wie oft bisher ein Sprung zu Label Nummer <j> entdeckt wurde. Da wir einen neuen Sprung entdeckten, wird dieser Tabelleneintrag um 1 erhöht (*LZahl(j) = LZahl(j) + 1*).

<LTab(X, Y)> enthält zu jedem Label 1 bis X eine Liste aller Zeilen 1 bis Y, in denen es in einem Sprungbefehl auftaucht. Da wir einen neuen Sprung zum Label Nummer <j> entdeckt haben, vermerken wir die Nummer der aktuellen Zeile am Ende der Liste des j-ten Labels, als Eintrag (*LTab(j, LZahl(j)) = i*).

Wie bereits erläutert, ist die Labelnummer <j> um eins größer als <LGesamt>, wenn das entdeckte Label zum erstenmal auftaucht. In diesem Fall wird <LGesamt> um eins erhöht und das neu entdeckte Label <L$> in der Labeltabelle als – vorläufig – letztes Element eingetragen (*If j > LGesamt Then LGesamt = j: L$(LGesamt) = L$*).

Die aktuelle Zeile ist vollständig behandelt. Theoretisch könnte CONVERT nun zu SEARCHJUMP verzweigen, um nach dem nächsten Sprung in Zeile <i> zu suchen. Mehrere Sprünge in einer Zeile sind keineswegs ungewöhnlich:

```
IF A = 1 THEN GOTO JUMP1 ELSE GOTO JUMP2
```

Dabei würden wir jedoch eine »Kleinigkeit« übersehen. Der Zeilennummer nach einem Sprungbefehl kann ohne weiteren Sprungbefehl sofort wieder ein Sprung folgen:

```
ON I GOTO  100,200,300
ON I GOSUB 100,200,300
```

Daher wird als neue aktuelle Position <Start> das Ende (<Ende>) des aktuellen Labels definiert und zu SEARCHNR verzweigt, wenn das Zeilenende noch nicht erreicht ist.

```
Jump:
    Start = Ende
    If Start <> Len(Line$(i)) Then Goto SearchNr
NextLine:
  Next i
```

SEARCHNR wird ab der aktuellen Position nach dem nächsten Nicht-Leerzeichen suchen (*Call SearchOther(Line$(i), Terminator$, Start) 'Zeilennummer?*). Wenn bis zum Zeilenende kein Nicht-Trennzeichen folgt, ist die Zeile behandelt und SEARCHNR verzweigt zu NEXTLINE, dem Schleifenende:

```
If Start = 0 Then Goto NextLine
```

Folgt keine Ziffer als nächstes Nicht-Trennzeichen, liegt auch kein »Mehrfachsprung« mit ON..GOTO oder ON..GOSUB vor. Die Analyse der Zeile geht ab der aktuellen Position <Start> weiter:

```
If Instr(Ziffer$, Mid$(Line$(i), Start, 1)) = 0 Then Goto SearchJump
```

Sind dagegen beide Bedingungen nicht erfüllt, können wir das so interpretieren: Ab <Start> gibt es ein Nicht-Trennzeichen und das erste Nicht-Trennzeichen ist eine Ziffer. Also folgt der gerade behandelten Zeilennummer sofort eine weitere. Da weder zu NEXTLINE noch zu SEARCHJUMP verzweigt wird, geht es mit dem Programmabschnitt nach »Zeilennummer ermitteln« weiter mit »Zeilennummer in Labeltabelle speichern«.

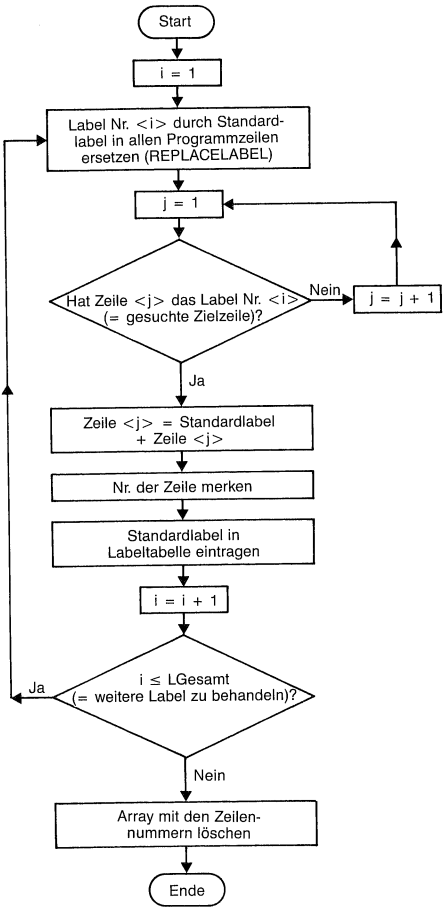


Bild 21 *Standardlabel einsetzen*

Die in <Line$(..)> enthaltene Programmzeile ist nun vollständig analysiert. Die verschiedenen Tabellen enthalten die »Label« (=Zeilennummern der Zielzeilen), die Labelanzahl, die Nummern der Programmzeilen, in denen sich ein Sprung zu einem Label befindet und die Anzahl der Sprünge zu jedem Label.

Wir ersetzen jedes Label in <L$..)> im gesamten Programmtext durch Standardlabel (»Label«<, »Label2« etc.).

```
'        *** Standardlabel einsetzen ***
  For i=1 to LGesamt
    LNeu$ = "Labe" + Str$(i): Mid$(LNeu$, 5, 1) = "l"
    Call ReplaceLabel(i, LNeu$)
    j = 1
```

```
    While LineNr$(j) <> L$(i)
      j = j + 1
    Wend
    Line$(j) = LNeu$ + ": " + Line$(j)
    LLine(i) = j
    L$(i) = LNeu$
  Next i
  Erase LineNr$   'Zeilennummern-Array wird nicht mehr gebraucht
End Sub
```

Die Schleifenvariable <i> gibt die aktuelle Labelnummer an. <LNeu$> ist das jeweilige Standardlabel und wird aus dem Wort »Label« und der Labelnummer gebildet.

Die aufgerufene Prozedur REPLACELABEL ersetzt das Label Nummer <i> im gesamten Text (nach jedem Sprungbefehl) durch das übergebene neue Label <LNeu$>. Damit ist das aktuelle Label allerdings noch nicht vollständig behandelt.

Es muß zusätzlich am Anfang der angesprungenen Programmzeile eingefügt werden:

```
Label1: PRINT "Dies ist ein Test"
```

Beim Einlesen des Programmtextes speicherten wir in <LineNr$(i)> die einzelnen Zeilennummern. Dieses Array durchsuchen wir nun in einer WHILE-Schleife solange, bis das alte Label (die angesprungene Zeilennummer) <L$(i)> gefunden ist.

<j> gibt uns den Index der angesprungenen Programmzeile im Array <Line$(..)> an. Am Anfang dieser Zeile wird das Standardlabel <Lneu$>, ein Doppelpunkt und zur optischen Trennung noch ein Leerzeichen eingefügt (*Line$(j) = LNeu$ + ": " + Line$(j)*).

Im Array <LLine(..)> wird der Index der Programmzeile gespeichert, vor der sich das aktuelle Label Nummer <i> befindet (*LLine(i) = j*).

Und im Labelarray <L$(..)> wird das alte Label Nummer <i> (die Zeilennummer) durch das erzeugte Standardlabel ersetzt (*L$(i) = LNeu$*).

Auf diese Weise werden alle im Programmtext vorkommenden Label 1 bis <LGesamt> behandelt.

Das Array <LineNr$(..)> enthielt die Zeilennummern aller(!) Programmzeilen. Dieses Array benötigten wir gerade, um den Index einer angesprungenen Programmzeile herauszubekommen. Inzwischen enthält jedoch <LLine(..)> für jedes Label den Index der angesprungenen Zeile. Da wir <LineNr$(..)> nicht mehr benötigen, löschen wir dieses Array (*Erase LineNr$*). ERASE löscht nicht nur die Variableninhalte, sondern gibt den belegten Speicher wieder frei.

Die Prozedur LOADFILE ist beendet und der am Anfang dieses Kapitels gezeigte Programmausschnitt sieht inzwischen so aus:

```
.....
.....
Label6:   X=INSTR(J,PZ$(I),"THEN"): REM 'then'  ab <j> vorhanden?
  Y=INSTR(J,PZ$(I),"GOTO"): REM 'goto'  ab <j> vorhanden?
  Z=INSTR(J,PZ$(I),"GOSUB"):REM 'gosub' ab <j> vorhanden?
  IF X=Ø AND Y=Ø AND Z=Ø THEN GOTO Label2:REM kein sprungbefehl
  :
Label5:   GOSUB Label3:REM bis zu erstem space vortasten
  GOSUB Label4:REM folgende spaces überlesen
  :
  IF J=Y OR J=Z THEN GOTO Label5:REM 'then goto'/'then gosub'
  IF INSTR(ZIFF$,MID$(PZ$(I),J,1))=Ø THEN GOTO Label6
  :
Label8:   GOSUB Label7:ZN$(NR)=ZN$:REM zeilennummer lesen
  :
  NR=NR+1
  IF J=LEN(PZ$(I)) THEN GOTO Label2
  IF MID$(PZ$(I),J,1)="," THEN J=J+1:GOTO Label8 ELSE GOTO Label6
Label2: NEXT I
.....
.....
```

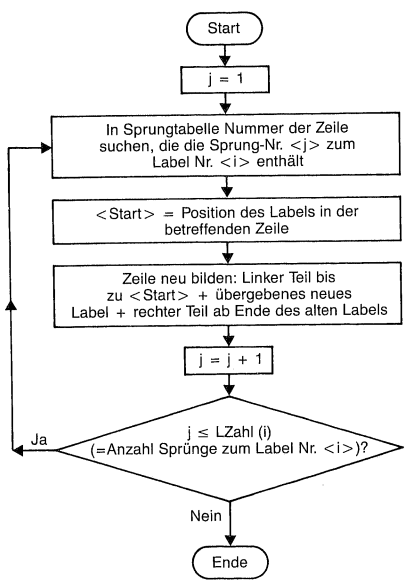


Bild 22 *Ersetzen eines Labels (REPLACELABEL)*

Die Erläuterung von REPLACELABEL steht noch aus. Diese Prozedur werden wir noch benötigen, um im gesamten Programmtext ein Standardlabel durch ein vom Benutzer gewähltes zu ersetzen.

Die Prozedur erwartet zwei Parameter: die Nummer des zu ersetzenden Label und das neue Label. Wie wir inzwischen wissen, ist die Labelnummer ein Zeiger auf die Arrays <L$(..)> (Tabelle der Label) und <LZahl(..)> (Anzahl der Sprünge zu jedem Label). Außerdem ist die Nummer ein Zeiger auf den ersten Index von <LTab(.., ..)> (Zeilennummern aller Sprünge zu den einzelnen Label).

```
'       ReplaceLabel
'       ------------
'       Funktion: Ersetzt ein Label im gesamten Programmtext
'       Hin     : i     : Nr. des zu ersetzenden Labels (in <L$(..)>)
'                 LNeu$ : Neues Label

Sub ReplaceLabel(i, LNeu$) Static
  For j=1 to LZahl(i)
    Start = Instr(Line$(LTab(i, j)), L$(i))
    Line$(LTab(i, j)) = Left$(Line$(LTab(i, j)), Start - 1) +_
                        LNeu$ + Mid$(Line$(LTab(i, j)), Start +_
                        Len(L$(i)))
  Next j
End Sub
```

Die Schleifenvariable <j> läuft von 1 bis zu <LZahl(i)>, der Nummer des letzten Sprungs zum Label Nummer <i>. <LTab(i, j)> enthält den Index der Zeile, in der sich der »j-te« Sprung zum »i-ten« Label befindet.

Mit INSTR wird die Position des Labels innerhalb der aktuellen Zeile gesucht und <Start> zugewiesen. Die betreffende Zeile <Line$(LTab(i, j))> wird neu zusammengesetzt, und zwar aus

- dem linken Teil bis zum alten Label
- dem übergebenen neuen Label <LNeu$>
- dem rechten Teil ab dem Ende des neuen Label

Beispiel: Mit »Sort« als neuem Label wird

`IF A=1 THEN GOTO Label1` ersetzt durch `IF A=1 THEN GOTO Sort`

Dieses Ersetzen wird für alle in der Tabelle <LTab(.., ..)> enthaltenen Zeilen mit Sprüngen zum Label Nummer <i> wiederholt.

Programmtext speichern (SAVEFILE)

Keine Angst. SAVEFILE ist auch nicht annähernd vergleichbar mit LOADFILE. Diese Prozedur speichert den konvertierten Programmtext in der vom Benutzer angegebenen Datei; mehr macht sie nicht.

```
'       SaveFile
'       --------
'       Funktion: Speichert Programmdatei ab
'                 (<Line$(1)> bis <Line$(Lines))>)


Sub SaveFile Static

'       * Dateiname ? *
  Text$(1) = "Abbrechen(ESC) oder Bestätigen(RETURN)"
  Call StandInput(1, "Dateiname", File$, 4Ø, Alpha$, zEsc$ + zReturn$,_
  Last$)
  If Last$ = zEsc$ Then Exit Sub

'       * Datei speichern *
  F$ = File$: If Instr(F$, ".") = Ø Then F$ = F$ + ".bas"
  Open F$ For Output As #1
  For i=1 to LineZahl
    Print #1, Line$(i)
  Next i
  Close #1
End Sub
```

Wie sie das macht, ist dem Beginn von LOADFILE sehr ähnlich: Der Benutzer wird auf die gleiche Weise mit Hilfe von STANDINPUT nach einem Datenamen gefragt. Wenn er die Eingabe mit ESC beendet, wird die Prozedur vorzeitig abgebrochen und er kehrt ins Hauptmenü zurück.

Sonst wird der angegebene Dateiname <File$> in die Variable <F$> kopiert und wenn nötig, die Erweiterung ».bas« angehängt. Die Datei <F$> wird geöffnet und SAVEFILE schreibt der Reihe nach alle Programmzeilen <Line$(1)> bis <Line$(LineZahl)> in die Datei.

Liste der Label (LLISTE)

Die Pull-down-Menüs enthalten das Kommando »Liste der Label«. Diese Funktion soll Ihnen einen Überblick über die im Programmtext vorhandenen Label verschaffen. Diese Liste benötigen Sie vor allem bei der individuellen Umbenennung eines Labels. Es wäre doch peinlich, wenn Sie das Label »Label46« in »Sort« umbenennen, obwohl Sie bereits »Label1« diesen neuen Namen gaben.

```
'       LListe
'       ------
'       Funktion: Gibt alle Labels aus

Sub LListe Static
  Call ClearScreen(2, 25)
  Locate 3, 1
  For i=1 to LGesamt
    Print L$(i),
  Next i
  Text$(1) = "Drücken Sie eine beliebige Taste"
  Call Warten(1, Key$, Invers)
End Sub
```

LLISTE löscht mit CLEARSCREEN alle Zeilen unterhalb der Menüleiste. Der Cursor wird auf den Beginn von Zeile 3 gesetzt und ab dieser Position alle in der Tabelle <L$(..)> enthaltenen Label ausgegeben.

```
ErrorHandling Initialisierung              PullGround    PushGround
Label5        Label6        Label7         Label8        Label9
Label10       Label11       Label12        Label13       Label14
Label15       Label16       Label17        Label18       Label19
Label20       Label21       Label22        Label23       Label24
Label25       Label26       Label27        Label28       Label29
Label30       Label31       Label32        Label33       Label34
Label35       Label36       Label37        Label38       Label39
Label40       Label41       Label42        Label43       Label44
Label45       Label46       Label47        Label48       Label49
Label50       Label51       Label52        Label53       Label54
Label55       Label56       Label57        Label58       Label59
Label60       Label61       Label62        Label63       Label64
```

WARTEN wird aufgerufen und nach einem Tastendruck des Benutzers zum Menü zurückgekehrt.

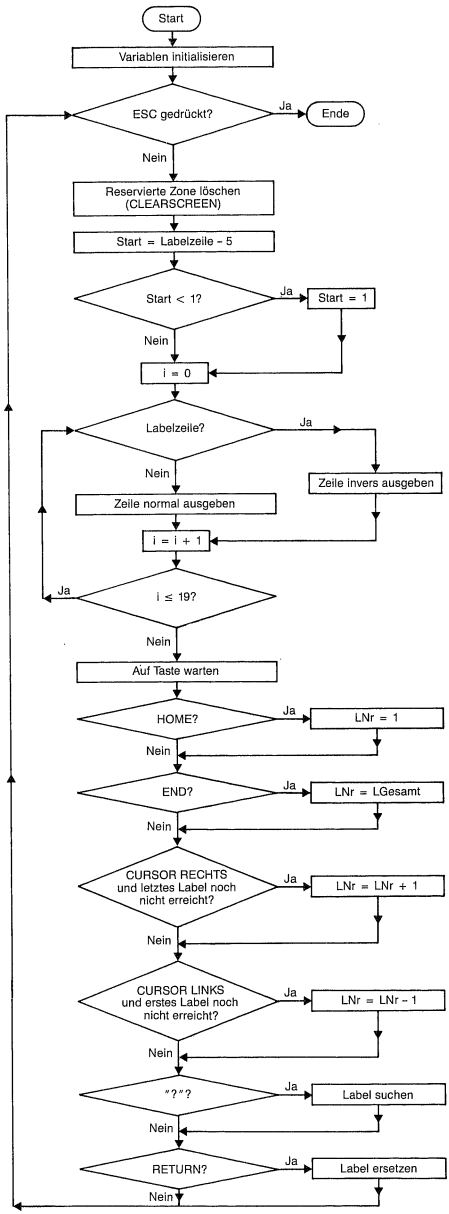


Bild 23 *Editieren der Label (LEDIT)*

LEDIT ist mal wieder eine umfangreichere Prozedur. Sie wird aufgerufen, wenn der Benutzer das Menükommando »Editieren« wählt.

LEDIT erlaubt Ihnen die Vergabe individueller Labelnamen und das so komfortabel wie möglich. Label sollten die Funktion des betreffenden Programmteils wiedergeben. Um ein Label durch ein anderes zu ersetzen, müssen Sie somit den Programmtext sehen.

LEDIT zeigt zu jedem Label die zugehörige »Umgebung«. Nach dem Aufruf dieser Prozedur erscheint auf dem Bildschirm die Zeile mit dem ersten Standardlabel »Label1«. Diese »Labelzeile« wird invers hervorgehoben. Darüber sehen Sie die vorhergehenden fünf Zeilen des Programmtextes und darunter die nächsten 15 Zeilen.

```
PUT #1,RECORD:                          REM kennzeichnen
RETURN
:
:
REM *** fehlerbehandlung ***
Label1: A$="Fehler Nummer"+STR$(ERR)+" aufgetreten!"
IF ERR=54 THEN A$="Falscher Dateityp!"
IF ERR=64 THEN A$="Unzulässiger Dateiname!"
IF ERR=68 THEN A$="Drucker anschliessen/einschalten!"
IF ERR=61 THEN A$="Diskette/Festplatte voll!"
IF ERR=71 THEN A$="Diskette einlegen/Klappe schliessen!"
IF ERR=70 THEN A$="Schreibschutz entfernen!"
IF ERR=53 THEN A$="Diese Datei existiert nicht!"
IF ERR=7  THEN A$="Der Hauptspeicher ist voll!"
IF ERR=27 THEN A$="Papier kontrollieren!"
IF ERR=76 THEN A$="Dieser Pfad existiert nicht!"
IF ERR=67 THEN A$="Inhaltsverzeichnis voll!"
:
FRAGE$=A$+"  (weiter mit ESC)":            REM laden und
E$="":CHAR$="":LENMAX=0:ENDE$=ESC$:       REM auf eine
```

Mit den Tasten CURSOR RECHTS und CURSOR LINKS blättern Sie zum nächsten (»Label2«) beziehungsweise vorhergehenden Label (das es in diesem Fall nicht gibt).

Mit HOME und END springen Sie direkt zum ersten Label (Label Nummer 1) beziehungsweise zum letzten (Label Nummer <LGesamt>). Immer sehen Sie außer der Labelzeile die Programmumgebung.

Sie können nun das gerade angezeigte Standardlabel durch ein sinnvolleres ersetzen. LEDIT bietet auch eine »Label-Suchfunktion«. Beispiel: Sie wollen im nachhinein prüfen, ob der von Ihnen vergebene Labelname »QuickSort« wirklich zutrifft oder Sie nicht versehentlich eine BubbleSort-Routine mit »QuickSort« benannt haben.

Dann suchen Sie einfach nach dem Label »QuickSort«. LEDIT springt unmittelbar zum Label und zeigt Ihnen die Programmumgebung. Sie müssen

sich daher nicht unbedingt immer mit der Blätterfunktion »durchwühlen«. Wenn Sie den Labelnamen kennen, ist Suchen vorzuziehen.

```
'       LEdit
'       -----
'       Funktion: Editieren der Programmdatei ('Labelweise' blättern/
'                 Label suchen/Label ersetzen)


Sub LEdit Static

'       * Initialisierung *
  LNr  = 1
  ExitFlag = False
  Text$(1) = "ESC: Zum Menü?: Label suchen RETURN: Label ersetzen"
  Text$(2) = "RECHTS: nächstes L. LINKS: voriges L. HOME: erstes L.  END:"_
  "letztes L."
```

Die mit 1 initialisierte Variable <LNr> gibt die Nummer des aktuellen Labels an. Die Texte <Text$(1)> und <Text$(2)> werden später am unteren Bildschirmrand als Hilfe für Sie zur Programmbedienung ausgegeben.

In der folgenden Schleife wird die Zeile, die das aktuelle Label enthält und das »Umfeld« ausgegeben, auf eine Taste gewartet und das jeweilige Kommando ausgeführt.

```
'       * Ausgabe des aktuellen Labels mit Umgebung *
  Key$ = ""
  While Key$ <> zEsc$
    Call ClearScreen(2, 21)
    Start = LLine(Lnr) - 5: If Start < 1 Then Start = 1
    For i=Ø to 19
      Attribut = Normal
      If Start + i = LLine(LNr) Then Attribut = Invers
      Call PrintF(1, i + 2, Line$(Start + i), Attribut)
    Next i
    Call Warten(2, Key$, Invers)
```

CLEARSCREEN löscht die Zeilen 2 bis 21. Zeile 1 enthält die Menüleiste. Das Löschen der Zeilen 22 bis 25 ist überflüssig, da in diesen Zeilen nach jedem Schleifendurchgang die Benutzerinformationen neu ausgegeben werden.

<LLine(LNr)> enthält den Index der Zeile, in der sich das aktuelle Label <Lnr> befindet. <Start> wird dieser Index minus 5 zugewiesen, da die Ausgabe mit der fünften Zeile vor der aktuellen Labelzeile beginnt. Wenn der Index kleiner als 1 ist, wird <Start> der Wert 1 zugewiesen (Ausgabe ab der ersten Programmzeile).

Die Zeilen <Line$(Start)> bis <Line$(Start + 19)> werden bis auf die Labelzeile <Line$(LLine(LNr))> normal dargestellt. Die Labelzeile hebt LEDIT invers hervor.

Nun wird WARTEN aufgerufen. Diese Prozedur gibt die zuvor definierten Infotexte aus, wartet auf eine Taste und übergibt sie in <Key$>.

```
'       * Unkritische Tasten behandeln *
    If Key$ = zHome$ Then LNr = 1
    If Key$ = zEnd$  Then LNr = LGesamt
    If Key$ = zRight$ And LNr < LGesamt Then LNr = LNr + 1
    If Key$ = zLeft$  And LNr > 1       Then LNr = LNr - 1
```

HOME »springt« zum ersten Label. <LNr> wird der Wert 1 zugewiesen. Im nächsten Schleifendurchgang gibt LEDIT die Zeile aus, die Label 1 enthält (plus Umgebung!).

END »zeigt« das letzte Label Nummer <LGesamt>, CURSOR RECHTS das nächste und CURSOR LINKS das dem aktuellen Label vorhergehende.

```
'       * Label suchen *
    If Key$ = "?" Then
      L$ = ""
      Call StandInput(0, "Suchen nach Label", L$, 20,_
                      Alpha$, zEsc$ + zReturn$, Last$)
      If Last$ = zReturn$ Then
        For i=1 to LGesamt
          If L$(i) = L$ Then LNr = i
        Next i
      End If
    End If
```

Die Taste »?« leitet die Labelsuche ein. STANDINPUT fordert Sie zur Eingabe des gesuchten Labels auf. Wenn Sie die Eingabe mit RETURN beenden, wird nun das Label <L$> in der Labeltabelle <L$(..)> gesucht.

Findet LEDIT Ihr gesuchtes Label, wird <LNr> die Labelnummer zugewiesen und im nächsten Durchgang »Ihr« Label mit Umgebung gezeigt.

```
'       * Label ersetzen *
    If Key$ = zReturn$ Then
      LNeu$ = ""
      Call StandInput(0, "'" + L$(LNr) + "' ersetzen durch", LNeu$, 20,_
                      Alpha$, zEsc$ + zReturn$, Last$)
      If Last$ = zReturn$ Then
        For i=1 to LZahl(LNr)
          Start = Instr(Line$(LTab(LNr, i)), L$(LNr))
          Ende  = Start + Len(L$(LNr))
          Line$(LTab(LNr, i)) = Left$(Line$(LTab(LNr, i)), Start - 1)_
                                + LNeu$ + Mid$(Line$(LTab(LNr, i)), Ende)
        Next i
```

Das Ersetzen eines Labels ist etwas schwieriger. Ersetzt wird immer das gerade gezeigte Label. STANDINPUT fordert Sie auf, einen neuen Namen zu vergeben (<LNeu$>).

Das alte Label ist nun in allen <LZahl(LNr)> Programmzeilen, die einen Sprung zu diesem Label enthalten, durch <LNeu$> zu ersetzen. Die Indizes dieser Zeilen enthält <LTab(.., ..)>. <Line$(LTab(LNr, i))> ist die Programmzeile, in der sich der i-te Sprung zum Label Nummer <LNr> befindet.

INSTR weist <Start> die Position des alten Labels in dieser Zeile zu (*Start = Instr(Line$(LTab(LNr, i)), L$(LNr))*). Die Position des Labelendes ist durch die Startposition plus der Labellänge definiert (*Ende = Start + Len(L$(LNr))*).

Die Zeile wird auf die schon gewohnte Weise neu zusammengesetzt: Linker Teil bis zum alten Label plus neues Label <LNeu$> plus rechter Teil ab dem Ende des alten Labels.

Auf die gleiche Weise werden alle anderen Zeilen behandelt, die einen Sprung zum aktuellen Label enthalten. Das alte Label wird immer durch <LNeu$> ersetzt.

Nun ersetzt LEDIT auch in der angesprungenen Zeile Nummer <LLine(LNr)> das alte Label und aktualisiert die Labeltabelle <L$(..)>.

```
        Line$(LLine(LNr)) = LNeu$ + Mid$(Line$(LLine(LNr)),
 Len(L$(LNr)) + 1)
        L$(LNr) = LNeu$
      End If
    End If
  Wend
End Sub
```

Die Tabelle <LLine(..)> enthält die Indizes aller Zielzeilen (Label am Zeilenanfang). <Line$(LLine(LNr))> ist demnach die Zielzeile, an deren Anfang sich das aktuelle Label Nummer <LNr> befindet. Das alte Label am Zeilenanfang wird durch <LNeu$> ersetzt.

Zum Schluß wird in der Labeltabelle <L$(..)> das alte Label Nummer <LNr> durch das neue Label ersetzt. Die Funktion »Label ersetzen« ist beendet.

Nun kennen Sie alle Funktionen von LEDIT. LEDIT verlassen Sie mit der ESC-Taste. <Key$> enthält die zuletzt gedrückte Taste. Die Schleifenbedingung *While Key$ = zEsc$* ist nicht mehr erfüllt, wennn Sie ESC drücken. Die Prozedur LEDIT ist beendet und Sie kehren zum Menü zurück.

REM durch »'« ersetzen (REPLACEREM)

Mit REPLACEREM können Sie die herkömmlichen Kommentarbefehle (REM) durch das in QuickBASIC gleichwertige Zeichen »'« ersetzen.

```
'       ReplaceRem
'       ----------
'       Funktion: Ersetzt alle REM's durch '


Sub ReplaceRem Static

'       * Sicherheitsabfrage *
  A$ = "n"
  Text$(1) = "Beispiel: Aus <REM Kommentar> wird <' Kommentar>"
  Text$(2) = "Abbrechen(ESC) oder Bestätigen(RETURN)
  Call StandInput(2, "Sind Sie sicher (j/n) ?", A$, 1,_
                  "jn", zEsc$ + zReturn$, Last$)
  If Last$ = zEsc$ Or A$ <> "j" Then Exit Sub
```

Zur Sicherheit fragt Sie REPLACEREM vor der Ausführung, ob Sie wirklich sicher sind und zeigt Ihnen ein Beispiel. Gehen wir davon aus, daß Sie diese Funktion nicht versehentlich anwählten.

```
'       * Ersetzen *
  For i=1 To LineZahl
    Start = Instr(Line$(i), "REM ")
    If Start = Ø Then Start = Instr(Line$(i), "Rem ")
    If Start <> Ø Then Line$(i) = Left$(Line$(i), Start - 1) +_
                                  "'" + Mid$(Line$(i), Start + 3)
  Next i
End Sub
```

In einer Schleife werden alle Programmzeilen auf Kommentarbefehle abgesucht. Mit einer weiteren Option, die wir gleich kennenlernen, können Sie die GW-BASIC-Großschrift (*THEN GOTO*) durch die üblichere Groß-Klein-Schrift ersetzen (*Then Goto*).

In diesem Fall wäre die Suche nach großgeschriebenen Kommentarbefehlen natürlich erfolglos.

Daher sucht REPLACEREM nicht nur nach »REM«, sondern auch nach »Rem«. Enthält die aktuelle Zeile einen Kommentarbefehl, wird Sie neu zusammengesetzt aus: dem linken Teil bis zum Kommentar (bis <Start>), dem Zeichen »'« und dem rechten Teil ab dem Kommentarbefehl.

Dieses Spiel wiederholt sich für alle Programmzeilen 1 bis <LineZahl>. Mit individuellen Label und QuickBASIC-Kommentaren sieht ein Programm schon wesentlich »freundlicher« aus:

```
PUT #1,RECORD:                          ' kennzeichnen
RETURN
:
:
' *** fehlerbehandlung ***
ErrorHandling: A$="Fehler Nummer"+STR$(ERR)+" aufgetreten!"
IF ERR=54 THEN A$="Falscher Dateityp!"
IF ERR=64 THEN A$="Unzulässiger Dateiname!"
IF ERR=68 THEN A$="Drucker anschliessen/einschalten!"
IF ERR=61 THEN A$="Diskette/Festplatte voll!"
IF ERR=71 THEN A$="Diskette einlegen/Klappe schliessen!"
IF ERR=70 THEN A$="Schreibschutz entfernen!"
IF ERR=53 THEN A$="Diese Datei existiert nicht!"
IF ERR=7  THEN A$="Der Hauptspeicher ist voll!"
IF ERR=27 THEN A$="Papier kontrollieren!"
IF ERR=76 THEN A$="Dieser Pfad existiert nicht!"
IF ERR=67 THEN A$="Inhaltsverzeichnis voll!"
:
FRAGE$=A$+"  (weiter mit ESC)":            ' laden und
E$="":CHAR$="":LENMAX=0:ENDE$=ESC$:       ' auf eine
```

Groß-Klein-Schrift (GROSSKLEIN)

GROSSKLEIN untersucht alle Zeilen auf großgeschriebene Anweisungen und Variablennamen. Der erste Buchstabe bleibt weiterhin großgeschrieben, der Rest der Anweisung wird in Kleinschrift gewandelt.

Auch GROSSKLEIN will vor Durchführung der nicht mehr rückgängig zu machenden Aktion von Ihnen eine Bestätigung.

```
'       GrossKlein
'       ----------
'       Funktion: Wandelt Befehlswörter und Variablennamen
'                 Groß-/Kleinschrift um

Sub GrossKlein Static

'       * Sicherheitsabfrage *
  A$ = "n"
  Text$(1) = "Beispiel: Aus <THEN GOTO> wird <Then Goto>"
  Text$(2) = "Abbrechen(ESC) oder Bestätigen(RETURN)
  Call StandInput(2, "Sind Sie sicher (j/n) ?", A$, 1,_
                  "jn", zEsc$ + zReturn$, Last$)
  If Last$ = zEsc$ Or A$ <> "j" Then Exit Sub
```

Diese Prozedur ist leider sehr langsam. In einer Schleife werden alle Programmzeilen behandelt. Und in einer inneren Schleife tastet sich GROSSKLEIN »zeichenweise« in der aktuellen Zeile voran.

```
'       * Schreibweise wandeln *
  UpChars$ = "QWERTZUIOPASDFGHJKLYXCVBNM"   'Alle Großbuchstaben
  For i=1 to LineZahl
    For Start=1 to Len(Line$(i))
      If Instr(UpChars$, (Mid$(Line$(i), Start, 1))) <> 0 Then
        Ende = Start
```

```
        Call SearchOther(Line$(i), UpChars$, Ende)
        If Ende - Start >= 2 Then
          S$ = Mid$(Line$(i), Start + 1, Ende - (Start + 1))
          Call UpLowCase(S$)
          Mid$(Line$(i), Start + 1) = S$
        End If
        Start = Ende
      End If
    Next Start
  Next i
End Sub
```

<Start> ist die aktuelle Position innerhalb der untersuchten Zeile Nummer <i>. Ist das Zeichen an dieser Position in <UpChar$> enthalten (=Großbuchstabe), beginnt an dieser Stelle ein Befehlswort oder ein Variablenname.

SEARCHOTHER sucht das Ende des Wortes und weist diese Endposition oder Variablen <Ende> zu. Wenn das Wort mindestens zwei Zeichen lang ist, (*If Ende - Start >= 2 Then*) wird es ab dem zweiten Zeichen <S$> zugewiesen. UPLOWCASE (in der Routinen-Sammlung) wandelt alle in <S$> enthaltenen Zeichen in Kleinbuchstaben um.

Der mittlere Teil der Zeile ab <Start> wird durch <S$> ersetzt. Wo zuvor Großbuchstaben waren, befinden sich nun die entsprechenden Kleinbuchstaben.

<Start> wird auf das Ende des behandelten Wortes gesetzt und die Suche geht weiter – die nächsten Befehlswörter/Variablennamen der aktuellen Zeile werden behandelt.

Und nun sieht das behandelte Programm fast schon wie ein echtes QuickBASIC-Programm aus.

```
' parameter hin: wcol/wrow: linke obere windowecke
'                mbreite  : menuebreite
'                mlength  : menuelänge
:
PrintMenue: Locate 2,Wcol(Menue)
Print "╓"+String$(Mbreite(Menue),"═")+"╖":' rahmen oben
For I=1 To Mlength(Menue):                 ' windowinnen-
  Locate ,Wcol(Menue):                     ' zeilen malen
  Print "│"+Menue$(Startstring(Menue)+I-1)+"│"
Next I
Locate ,Wcol(Menue)
Print "╙"+String$(Mbreite(Menue),"═")+"╜":' rahmen unten
Return
:
:
' *** neuer befehl angewählt ***
' funktion: normalisiert aktuellen befehl, invertiert
'           neu angewählten befehl
```

Inhaltsverzeichnis ausgeben (DIRECTORY)

Uns fehlen nur noch drei winzige Prozeduren, dann haben wir CONVERT komplett abgehandelt. DIRECTORY gibt das aktuelle Inhaltsverzeichnis aus und verdient eigentlich keine weitere Erwähnung.

```
'       Directory
'       ---------
'       Funktion: Directory ausgeben und auf Taste warten

Sub Directory Static
  Call ClearScreen(2, 25)
  Locate 3, 1
  Files
  Text$(1) = "Drücken Sie eine beliebige Taste"
  Call Warten(1, Key$, Invers)
End Sub
```

Aktuelles Verzeichnis wechseln (EDITDIR)

EDITDIR ist schon erheblich interessanter und das nur wegen einer Variablen. <Dir$> ist eine »statische« Variable.

Normalerweise sind alle in einer Prozedur verwendeten Variablen »dynamische Variablen« und werden beim Verlassen der Prozedur gelöscht.

Mit dem Attribut STATIC können Sie beliebige Variablen als »statisch« deklarieren. Statische Variablen werden nicht gelöscht. Sie behalten Ihren letzten Wert!

```
'       EditDir
'       -------
'       Funktion: Aktuelles Verzeichnis wechseln

Sub EditDir Static
  Static Dir$
  Text$(1) = "Abbrechen(ESC) oder Bestätigen(RETURN)"
  Call StandInput(1, "Aktuelles Verzeichnis", Dir$, 30,_
                  Alpha$, zEsc$ + zReturn$, Last$)
  If Last$ = zReturn$ Then Chdir Dir$
End Sub
```

EDITDIR fragt Sie nach dem gewünschten aktuellen Verzeichnis. Der aktuelle Inhalt von <Dir$> wird vorgegeben. CHDIR (»change directory«) wechselt das Verzeichnis. Sie dürfen beim Verzeichniswechsel wie gewohnt beliebige Pfade angeben, zum Beispiel »c:\qb\demoprog«.

Da <Dir$> eine statische Variable ist, wird Ihnen beim nächsten Aufruf dieser Programmfunktion Ihre letzte Eingabe vorgegeben (»c:\qb\demoprog«). Sie ist unverändert in <Dir$> enthalten! Der Vorteil: Wenn sich nur der letzte Teil des Pfades ändert, müssen Sie nicht erneut alle anderen Verzeichnisse angeben.

Verlassen des Programms (ENDPROG)

ENDPROG ist mal wieder eine »triviale« Prozedur. Sie fragt, ob Sie sich auch wirklich sicher sind und – wenn ja – beendet sie das Programm.

```
'       EndProg
'       -------
'       Funktion: Programm nach Sicherheitsabfrage beenden

Sub EndProg Static
  A$ = "n"
  Call StandInput(Ø, "Sind Sie sicher (j/n) ?", A$, 1,_
                  "jn", zEsc$ + zReturn$, Last$)
  If Last$ = zReturn$ And A$ = "j" Then End
End Sub
```

8.5 Listing des Konvertierungsprogramms (CONVERT.BAS)

```
'       ************************************************
'       * Convert: GW-BASIC nach QuickBASIC konvertieren *
'       ************************************************

'       (C) Said Baloui, 1987


REM $INCLUDE: 'comdef.bas'

Common Shared Line$(1)          'Programmzeilen (dynamisches Array)
Common Shared L$(1), LLine(1)   'Label und Nummer der zugehörigen Zeile
Common Shared LZahl(1)          'Anzahl der Sprünge zu diesem Label
Common Shared LTab(2)           'Zeilen, in denen sich die Sprünge zu
                                'einem bestimmten Label befinden
Common Shared Text$(1)          'Verschiedene Texte, z.B. Fehlermeldungen
Common Shared LGesamt           'Gesamtanzahl der Label
Common Shared LineZahl          'Anzahl der Programmzeilen
Common Shared File$, Dir$       'Dateiname und aktuelles Verzeichnis
Common Shared LineMax, JumpMax  'Max. Anzahl an Sprüngen

LineMax = 1ØØØ: LMax = 1ØØ: JumpMax = 5Ø  'Ändern: Verkleinern, wenn Ihr
Dim L$(LMax), LLine(LMax), LZahl(LMax)    'RAM zu klein ist, vergrößern,
Dim LTab(LMax, JumpMax)                   'wenn die Arrays für Ihre GW-
Dim Text$(1Ø)                             'BASIC-Programme zu klein sind
File$ = ""

REM $INCLUDE: 'init.bas'
'-----------------------------------------------------------------------
```

```
'       Hauptprogramm
'       -------------
On Error Goto ErrorHandling

Header$   = "      Datei  Editieren  Sonstiges
"
Menue$(1) = "Laden#Speichern#"
Menue$(2) = "Liste#Editieren#REM ersetzen#Gross/Klein#"
Menue$(3) = "Directory#Zugriffspfad#-#Programm verlassen#"

MainLoop:Call PullDown(Menue, Befehl)
If Menue = 1 Then
  If Befehl = 1 Then Call LoadFile
  If Befehl = 2 And LineZahl <> Ø Then Call SaveFile
ElseIf Menue = 2 And LineZahl <> Ø Then
  If Befehl = 1 Then Call LListe
  If Befehl = 2 Then Call LEdit
  If Befehl = 3 Then Call ReplaceRem
  If Befehl = 4 Then Call GrossKlein
ElseIf Menue = 3 Then
  If Befehl = 1 Then Call Directory
  If Befehl = 2 Then Call EditDir
  If Befehl = 3 Then Call EndProg
End If
Goto MainLoop
'-----------------------------------------------------------------------------

'       Meldung ausgeben/Auf Taste warten
'       ---------------------------------
'       Funktion: Gibt Meldung in Window aus und wartet auf eine Taste
'       Hin     : Text$(..) : Meldung
'                 Texte     : Anzahl der Meldungen
'                 Attribut  : Darstellungsart (normal, invers etc.)
'       Zurück  : Key$      : Gedrückte Taste
'       Aufbau des Windows  :
'                             ┌──────────────────────┐
'                             │ Meldung 1 (Text$(1)) │
'                             │ Meldung 2 (Text$(2)) │
'                             │ Meldung 3 (Text$(3)) │
'                             └──────────────────────┘

Sub Warten(Texte, Key$, Attribut) Static

'       * Parameter des Gesamtwindows ermitteln *
  WLaenge = 2 + Texte
  WBreite = Ø
    For i=1 to Texte
      If Len(Text$(i)) > WBreite Then WBreite = Len(Text$(i))
    Next i
  WBreite = WBreite + 4
  WCol = 4Ø - Int(WBreite / 2)
  WRow = 26 - WLaenge

'       * Untergrund retten/Rahmen ausgeben *
  Call PushScreen(WCol, WRow, WBreite, WLaenge)
  Call Rahmen(WCol, WRow, WBreite, WLaenge, Attribut, 4)
```

```
'       * Texte ausgeben *
  For i=1 to Texte
    Call PrintF(WCol + 2, WRow + i, Text$(i), Attribut)
  Next i

'       * Auf Taste warten/Untergrund holen *
  Call Taste(Key$, Esc)
  Call PullScreen(WCol, WRow, WBreite, WLaenge)
End Sub
'-----------------------------------------------------------------------------

'       Standardeingabe
'       ---------------
'       Funktion: Übernimmt Eingabe in eigenem Window, das zeilenmittig
'                 zentriert wird
'       Hin     : Text$(..) : Kommentare, die unterhalb der Eingabe
'                             erscheinen
'                 Texte     : Anzahl zusätzlicher Kommentare
'                 Comment$  : Kommentar unmittelbar vor der Eingabe
'                 E$        : Vorgabe
'                 LenMax    : Maximale Eingabelänge
'                 Char$     : Eingabezeichen
'                 Back$     : Endezeichen
'       Zurück  : E$        : Eingabe
'                 Last$     : Endezeichen
'       Aufbau der Windows  :
'
'                             Kommentar  Eingabezone
'
'
'                             Zusatzkommentar (Text$(1))
'                             Zusatzkommentar (Text$(2))
'

Sub StandInput(Texte, Comment$, E$, LenMax, Char$, Back$, Last$) Static

'       * Parameter des Gesamtwindows ermitteln *
  WLaenge = 6 + Texte
  WBreite = Len(Comment$) + 2 + LenMax + 1
  For i=1 to Texte
    If Len(Text$(i)) > WBreite Then WBreite = Len(Text$(i))
  Next i
  WBreite = WBreite + 4
  WCol = 4Ø - Int(WBreite / 2)
  WRow = 2Ø - Wlaenge

'       * Untergrund retten/Rahmen ausgeben *
  Call PushScreen(WCol, WRow, WBreite, WLaenge)
  Call Rahmen(WCol, WRow, WBreite, WLaenge, Intensiv, 1)

'       * Parameter des Eingabewindows ermitteln *
  WInputCol    = WCol + 1 + Len(Comment$) + 2
  WInputRow    = WRow + 1
  WInputBreite = LenMax + 2
  WInputLaenge = 3
```

```
'       * Inhalt des Gesamtwindows ausgeben *
  Call Rahmen(WInputCol, WInputRow, WInputBreite, WInputLaenge, Normal, 3)
  Call PrintF(WCol + 2, WRow + 2, Comment$, Intensiv)
  For i=LBound(Text$) to LBound(Text$) + Texte - 1
    Call PrintF(WCol + 2, WRow + 4 + i, Text$(i), Normal)
  Next i

'       * Eingaberoutine aufrufen *
  Insert = False: PosAkt = 1
  Call LInput(E$, WInputCol + 1, WRow + 2, LenMax, Char$,_
              False, Back$, Insert, PosAkt, Last$, EditFlag)

'       * Untergrund holen *
  Call PullScreen(WCol, WRow, WBreite, WLaenge)
End Sub
'-----------------------------------------------------------------------------

'       LoadFile
'       --------
'       Funktion: Lädt Programmdatei, entfernt die Zeilennummern und
'                 ersetzt Sprungziele und Sprünge durch 'Standard-Label'
'                 (Label1, Label2 etc.)


Sub LoadFile Static

'       * Dateiname ? *
  Locate 3, 1: Files
  Text$(1) = "Abbrechen (ESC) oder Bestätigen (RETURN)"
  Call StandInput(1, "Dateiname", File$, 40, Alpha$, zEsc$ + zReturn$,_
  Last$)
  If Last$ = zEsc$ Then Exit Sub

'       * Vorbereitungen zum Einlesen *
  For i=1 to JumpMax
    LZahl(i) = 0
  Next i
  LineZahl = 0

'       * Anzahl Programmzeilen ermitteln/dynamische Arrays dimensionieren *
  F$ = File$: If Instr(F$, ".") = 0 Then F$ = F$ + ".bas"
  Open F$ For Input As #1
  While Not Eof(1)
    Line Input #1, A$
    LineZahl = LineZahl + 1
  Wend
  Close #1

  REM $DYNAMIC
  Redim Line$(LineZahl), LineNr$(LineZahl)

'       * Prog-Zeilen lesen, dabei Zeilennummern entfernen *
  Open F$ For Input As #1
  For i=1 to LineZahl
    Line Input #1,Line$(i)
    j = Instr(Line$(i), " ")
```

```
    LineNr$(i) = Left$(Line$(i), j - 1)
    Line$(i)   = Mid$(Line$(i), j + 1) + " "
  Next i
  Close #1


'       *** Label ermitteln ***
  Ziffer$ = "1234567890"
  LGesamt = 0
  For i=1 to LineZahl
    Start = 1

'       * Sprungbefehl suchen *
SearchJump:
    P1 = Instr(Start, Line$(i), "GOTO")
    P2 = Instr(Start, Line$(i), "GOSUB")
    P3 = Instr(Start, Line$(i), "RESUME")
    P4 = Instr(Start, Line$(i), "THEN")
    Min = Len(Line$(i))
    If P1 <> 0 Then Min = P1: L = 4
    If P2 <> 0 And P2 < Min Then Min = P2: L = 5
    If P3 <> 0 And P3 < Min Then Min = P3: L = 6
    If P4 <> 0 And P4 < Min Then Min = P4: L = 4
    If Min = Len(Line$(i)) Then Goto NextLine Else Start = Min

'       * Sonderbehandlung von THEN *
    If Start = P4 Then
      Start = Start + 4
      Call SearchOther(Line$(i), Terminator$, Start)    'Zeilennummer?
      If Instr(Ziffer$, Mid$(Line$(i), Start, 1)) <> 0 Then_
        Line$(i) = Left$(Line$(i), Start - 1) + "GOTO " +_
                   Mid$(Line$(i), Start)_
        Else Goto SearchJump
    End If
    Start = Start + L

'       * Zeilennummer ermitteln *
SearchNr:
    Call SearchOther(Line$(i), Terminator$, Start) 'Zeilennummer?
    If Start = 0 Then Goto NextLine
    If Instr(Ziffer$, Mid$(Line$(i), Start, 1)) = 0 Then Goto SearchJump

'       * Zeilennummer in Labeltabelle speichern *
    Ende = Start
    Call SearchOther(Line$(i), Ziffer$, Ende)
    L$ = Mid$(Line$(i), Start, Ende - Start)
    If L$ = "0" And Left$(Line$(1), 1) <> "0" Then Goto Jump
    j = 1
    While L$(j) <> L$ And j <= LGesamt
      j = j + 1
    Wend
    LZahl(j) = LZahl(j) + 1
    LTab(j, LZahl(j)) = i
    If j > LGesamt Then LGesamt = j: L$(LGesamt) = L$
Jump:
    Start = Ende
    If Start <> Len(Line$(i)) Then Goto SearchNr
```

```
NextLine:
  Next i


'       *** Standardlabel einsetzen ***
  For i=1 to LGesamt
    LNeu$ = "Labe" + Str$(i): Mid$(LNeu$, 5, 1) = "1"
    Call ReplaceLabel(i, LNeu$)
    j = 1
    While LineNr$(j) <> L$(i)
      j = j + 1
    Wend
    Line$(j) = LNeu$ + ": " + Line$(j)
    LLine(i) = j
    L$(i) = LNeu$
  Next i
  Erase LineNr$   'Zeilennummern-Array wird nicht mehr gebraucht
End Sub
'-----------------------------------------------------------------------------

'       ReplaceLabel
'       ------------
'       Funktion: Ersetzt ein Label im gesamten Programmtext
'       Hin     : i     : Nr. des zu ersetzenden Labels (in <L$(..)>)
'                 LNeu$ : Neues Label

Sub ReplaceLabel(i, LNeu$) Static
  For j=1 to LZahl(i)
    Start = Instr(Line$(LTab(i, j)), L$(i))
    Line$(LTab(i, j)) = Left$(Line$(LTab(i, j)), Start - 1) +_
                        LNeu$ + Mid$(Line$(LTab(i, j)), Start +_
                        Len(L$(i)))
  Next j
End Sub
'-----------------------------------------------------------------------------

'       SaveFile
'       --------
'       Funktion: Speichert Programmdatei ab
'                 (<Line$(1)> bis <Line$(Lines)>)


Sub SaveFile Static

'       * Dateiname ? *
  Text$(1) = "Abbrechen(ESC) oder Bestätigen(RETURN)"
  Call StandInput(1, "Dateiname", File$, 40, Alpha$, zEsc$ + zReturn$,-
   Last$)
  If Last$ = zEsc$ Then Exit Sub

'       * Datei speichern *
  F$ = File$: If Instr(F$, ".") = 0 Then F$ = F$ + ".bas"
  Open F$ For Output As #1
  For i=1 to LineZahl
    Print #1, Line$(i)
  Next i
  Close #1
End Sub
```

```
'-----------------------------------------------------------------------

'       ClearScreen
'       -----------
'       Funktion: Löscht angegebenen Zeilenbereich
'       Hin     : First : erste zu löschende Zeile
'                 Last  : letzte zu löschende Zeile

Sub ClearScreen(First, Last) Static
  For i=First to Last
    Call PrintF(1, i, Space$(8Ø), Normal)
  Next i
End Sub
'-----------------------------------------------------------------------

'       LListe
'       ------
'       Funktion: Gibt alle Labels aus

Sub LListe Static
  Call ClearScreen(2, 25)
  Locate 3, 1
  For i=1 to LGesamt
    Print L$(i),
  Next i
  Text$(1) = "Drücken Sie eine beliebige Taste"
  Call Warten(1, Key$, Invers)
End Sub
'-----------------------------------------------------------------------

'       LEdit
'       -----
'       Funktion: Editieren der Programmdatei ('Labelweise' blättern/
'                 Label suchen/Label ersetzen)


Sub LEdit Static

'       * Initialisierung *
  LNr   = 1
  ExitFlag = False
  Text$(1) = "ESC: Zum Menü?: Label suchen RETURN: Label ersetzen"
  Text$(2) = "RECHTS: nächstes L. LINKS: voriges L. HOME: erstes L. END:"_
   "letztes L."

'       * Ausgabe des aktuellen Labels mit Umgebung *
  Key$ = ""
  While Key$ <> zEsc$
    Call ClearScreen(2, 21)
    Start = LLine(Lnr) - 5: If Start < 1 Then Start = 1
    Ende  = 19: If Start + Ende > LineZahl Then Ende = LineZahl - Start
    For i=Ø to Ende
      Attribut = Normal
      If Start + i = LLine(LNr) Then Attribut = Invers
      Call PrintF(1, i + 2, Line$(Start + i), Attribut)
    Next i
    Call Warten(2, Key$, Invers)
```

```
'       * Unkritische Tasten behandeln *
    If Key$ = zHome$ Then LNr = 1
    If Key$ = zEnd$  Then LNr = LGesamt
    If Key$ = zRight$ And LNr < LGesamt Then LNr = LNr + 1
    If Key$ = zLeft$  And LNr > 1       Then LNr = LNr - 1

'       * Label suchen *
    If Key$ = "?" Then
      L$ = ""
      Call StandInput(Ø, "Suchen nach Label", L$, 2Ø,_
                      Alpha$, zEsc$ + zReturn$, Last$)
      If Last$ = zReturn$ Then
        For i=1 to LGesamt
          If L$(i) = L$ Then LNr = i
        Next i
      End If
    End If

'       * Label ersetzen *
    If Key$ = zReturn$ Then
      LNeu$ = ""
      Call StandInput(Ø, "'" + L$(LNr) + "' ersetzen durch", LNeu$, 2Ø,_
                      Alpha$, zEsc$ + zReturn$, Last$)
      If Last$ = zReturn$ Then
        For i=1 to LZahl(LNr)
          Start = Instr(Line$(LTab(LNr, i)), L$(LNr))
          Ende  = Start + Len(L$(LNr))
          Line$(LTab(LNr, i)) = Left$(Line$(LTab(LNr, i)), Start - 1)_
                                + LNeu$ + Mid$(Line$(LTab(LNr, i)), Ende)
        Next i
        Line$(LLine(LNr)) = LNeu$ + Mid$(Line$(LLine(LNr)),
                            Len(L$(LNr)) + 1)
        L$(LNr) = LNeu$
      End If
    End If
  Wend
End Sub
'-----------------------------------------------------------------------

'       ReplaceRem
'       ----------
'       Funktion: Ersetzt alle REM's durch '


Sub ReplaceRem Static

'       * Sicherheitsabfrage *
  A$ = "n"
  Text$(1) = "Beispiel: Aus <REM Kommentar> wird <' Kommentar>"
  Text$(2) = "Abbrechen(ESC) oder Bestätigen(RETURN)
  Call StandInput(2, "Sind Sie sicher (j/n) ?", A$, 1,_
                  "jn", zEsc$ + zReturn$, Last$)
  If Last$ = zEsc$ Or A$ <> "j" Then Exit Sub
```

```
'       * Ersetzen *
  For i=1 To LineZahl
    Start = Instr(Line$(i), "REM ")
    If Start = Ø Then Start = Instr(Line$(i), "Rem ")
    If Start <> Ø Then Line$(i) = Left$(Line$(i), Start - 1) +_
                                  "'" + Mid$(Line$(i), Start + 3)
  Next i
End Sub
'-----------------------------------------------------------------------

'       GrossKlein
'       ----------
'       Funktion: Wandelt Befehlswörter und Variablennamen
'                 Groß-/Kleinschrift um

Sub GrossKlein Static

'       * Sicherheitsabfrage *
  A$ = "n"
  Text$(1) = "Beispiel: Aus <THEN GOTO> wird <Then Goto>"
  Text$(2) = "Abbrechen(ESC) oder Bestätigen(RETURN)
  Call StandInput(2, "Sind Sie sicher (j/n) ?", A$, 1,_
                  "jn", zEsc$ + zReturn$, Last$)
  If Last$ = zEsc$ Or A$ <> "j" Then Exit Sub

'       * Schreibweise wandeln *
  UpChars$ = "QWERTZUIOPASDFGHJKLYXCVBNM"   'Alle Großbuchstaben
  For i=1 to LineZahl
    For Start=1 to Len(Line$(i))
      If Instr(UpChars$, (Mid$(Line$(i), Start, 1))) <> Ø Then
        Ende = Start
        Call SearchOther(Line$(i), UpChars$, Ende)
        If Ende - Start >= 2 Then
          S$ = Mid$(Line$(i), Start + 1, Ende - (Start + 1))
          Call UpLowCase(S$)
          Mid$(Line$(i), Start + 1) = S$
        End If
        Start = Ende
      End If
    Next Start
  Next i
End Sub
'-----------------------------------------------------------------------

'       Directory
'       ---------
'       Funktion: Directory ausgeben und auf Taste warten

Sub Directory Static
  Call ClearScreen(2, 25)
  Locate 3, 1
  Files
  Text$(1) = "Drücken Sie eine beliebige Taste"
  Call Warten(1, Key$, Invers)
End Sub
'-----------------------------------------------------------------------
```

```
'       EditDir
'       -------
'       Funktion: Aktuelles Verzeichnis wechseln

Sub EditDir Static
  Static Dir$
  Text$(1) = "Abbrechen(ESC) oder Bestätigen(RETURN)"
  Call StandInput(1, "Aktuelles Verzeichnis", Dir$, 30,_
                  Alpha$, zEsc$ + zReturn$, Last$)
  If Last$ = zReturn$ Then Chdir Dir$
End Sub
'-----------------------------------------------------------------------

'       EndProg
'       -------
'       Funktion: Programm nach Sicherheitsabfrage beenden

Sub EndProg Static
  A$ = "n"
  Call StandInput(0, "Sind Sie sicher (j/n) ?", A$, 1,_
                  "jn", zEsc$ + zReturn$, Last$)
  If Last$ = zReturn$ And A$ = "j" Then End
End Sub
'-----------------------------------------------------------------------

'       ErrorHandling
'       -------------
'       Funktion: Behandelt das Auftreten von Fehlern
'                 Fehler wird gemeldet und zum Menü verzweigt

ErrorHandling:
Text$(1) = "Fehler Nummer" + Str$(Err) + " aufgetreten"
If Err = 76 Then Text$(1) = "Dieses Verzeichnis existiert nicht"
If Err = 53 Then Text$(1) = "Diese Datei existiert im nicht im aktuellen"_
   "Verzeichnis"
Call Warten(1, Key$, Invers)
Resume MainLoop
```

8.6 Anwendung von CONVERT

In den folgenden Kapiteln zeige ich Ihnen, wie Sie Programme erstellen, die ohne Kompilation lauffähig sind. Bis dahin ist es jedoch ein weiter Weg. Und Sie wollen CONVERT wahrscheinlich sofort ausprobieren. Kein Problem, kompilieren Sie einfach wie gewohnt im Speicher.

Laden Sie QuickBASIC und unsere User-Library. Sie enthält bereits alle von CONVERT benötigten Module. Laden Sie nun CONVERT.BAS. Bevor Sie mit F5 oder CTRL+r die Kompilation starten, begeben Sie sich bitte in das Menü mit den Kompilieroptionen.

Aktivieren Sie die Option »On Error«. Und nun kompilieren Sie das Programm. Wenn alles klappt, sollten nach dem Starten die Pull-down-Menüs erscheinen.

Funktionsbeschreibung

- Laden: Laden eines im ASCII-Format gespeicherten GW-BASIC-Programms (Pfadangabe möglich).
- Speichern: Speichern des konvertierten Programmtextes (Pfadangabe möglich).
- Liste: Liste aller Label (Standard- und benutzerdefinierte Label).
- Editieren: Anzeigen der Label mit Umgebung; globales Ersetzen von Label; Suchen bestimmter Label.
- REM ersetzen: Globales Ersetzen von »REM« durch »'«.
- Groß-Klein: Die Standard-Großschrift von GW-BASIC bei Anweisungen und Variablennamen durch Groß-Klein-Schreibung ersetzen (erster Buchstabe groß, alle weiteren klein).
- Directory: Inhalt des aktuellen Verzeichnisses anzeigen.
- Zugriffspfad: Wechseln des aktuellen Verzeichnisses.

9

Lauffähige Programme mit LINK.EXE erzeugen

Bevor Sie weiterlesen: In den folgenden Kapiteln gehe ich auf verschiedene QuickBASIC-Systemdateien ein, auf BRUN20.LIB, BRUN20.EXE und BCOM20.LIB. Sollten Sie die Version 3.0 von QuickBASIC besitzen, können Sie lange nach diesen Dateien suchen.

Bei Ihnen ist die Zahl 20 im Dateinamen durch 30 ersetzt. Sie dürfen ensprechend die Dateien BRUN30.LIB, BRUN30.EXE und BCOM30.LIB verwenden!

CONVERT ist nach zahllosen unterschiedlich komplexen Prozeduren unser erstes Anwendungsprogramm. Dieses Programm werden wir so bearbeiten, daß anschließend kein Kompilieren mehr nötig ist, um CONVERT zu benutzen.

Aber zuvor sollten Sie – falls Sie es noch nicht getan haben – CONVERT endlich ausprobieren. Denken Sie daran: CONVERT benötigt unsere User-Library und muß mit der Option »On Error« kompiliert werden!

Sollte QuickBASIC die Kompilierung mit »Compilerspeicher nicht ausreichend« abbrechen, verkleinern Sie wie erläutert die Arraydimensionen (beziehungsweise entfernen Sie zuvor all Ihre speicherresidenten Programme).

9.1 Voraussetzungen zum Ablauf eines Programms

Vorab: Sie werden nun keinesfalls alle Einzelheiten über verschiedene Dateiformen wie ».OBJ-«, ».COM«- oder ».EXE«-Files erfahren.

Dafür sind spezielle DOS-Bücher zuständig. Uns interessieren momentan keine Details, sondern wie wir das immer wiederkehrende Kompilieren vermeiden können. Maschinensprache-Freaks bitte ich darum, sich bei den fol-

genden Erläuterungen das Lachen zu verkneifen. Dieses Buch behandelt BASIC.

Zuerst müssen wir uns darüber klar werden, welche Voraussetzungen eigentlich zur Ausführung eines Programms erfüllt sein müssen.

Kompilieren bedeutet Umwandeln eines Programmtextes in ein Maschinensprache-Programm. Um genau zu sein: Das »Kompilat«, die erzeugte Objektdatei, ist eigentlich nur eine Vorstufe auf dem Weg zum endgültigen Programm und muß noch »gelinkt« werden. Eine WHILE-Schleife sieht in einer Objektdatei grob vereinfacht so aus:

```
.....
.....
Sprung zu Label WHILE
  (Maschinensprache-Befehl)
  (Maschinensprache-Befehl)
Sprung zu Label WEND
.....
.....
```

Das Kompilat enthält keine BASIC-Anweisungen mehr, sondern entsprechende Maschinensprache-Befehle. Aber nicht überall werden BASIC-Anweisungen schon durch die zugehörigen Maschinenbefehle ersetzt.

Denn Anweisungen wie WHILE, PRINT oder WEND entsprechen nicht einfach einem Maschinenbefehl. Für die Ausführung solcher Befehle sind komplette Unterprogramme nötig. Und diese Programme befinden sich in den »Librarys« (Bibliotheken) von QuickBASIC. Beim Kompilieren setzt QuickBASIC an den entsprechenden Stellen einfach Label mit dem Namen der entsprechenden Programme ein. Die eigentlichen Maschinenroutinen zur Ausführung dieser Anweisungen werden erst beim »Linken« der Objektdateien integriert.

Stellen Sie sich eine Library so vor:

```
Label PRINT:   Befehl 1
               Befehl 2
               ...
               ...
               Befehl N
```

Die Library enthält Maschinensprache-Programme zur Ausführung unserer BASIC-Anweisungen. Jedes Programm ist mit einem entsprechenden Label gekennzeichnet. Es gibt also ein Programm GOTO, ein Programm PRINT und so weiter.

Beim »Linken« wird die Objektdatei analysiert. Das Linkprogramm erstellt eine Tabelle, die alle Aufrufe an die Library enthält. Es analysiert auch die Library und sucht in ihr nach den zugehörigen Label. Anschließend weiß der

Linker, welche Teile aus der Library wo in der Objektdatei benötigt werden. Er kann Objektdatei und Library »zusammenbinden« und die Label der Objektdatei durch den Aufruf dieser Programme ersetzen (durch »Sprünge« zu den entsprechenden Adressen, ähnlich GOTO 1234).

QuickBASIC verfügt über zwei Linkprogramme. Eines kennen Sie bereits, BUILDLIB. BUILDLIB bindet mehrere Objektdateien zu einer User-Library zusammen. Alle Objektdateien werden analysiert und ermittelt, wo welche Programmteile benötigt werden. In die erzeugte .EXE-Datei werden diese Programmteile eingebunden und die Label durch Sprünge zu den Adressen dieser Programmteile ersetzt. Wir erhalten ein echtes Maschinensprache-Programm in direkt vom Prozessor ausführbarer Form.

Im Gegensatz zur User-Library liegt unser Hauptprogramm CONVERT jedoch nicht in ausführbarer Form vor, sondern nur als Programmtext. Um CONVERT zu benutzen, müssen wir daher immer zuerst QuickBASIC (und die User-Library) laden und CONVERT.BAS kompilieren. Erst anschließend befinden sich beide Programmteile (unsere User-Library und das Hauptprogramm CONVERT) in ausführbarer Form im Speicher.

Unsere Programme allein genügen jedoch nicht zur Ausführung. CONVERT enthält schließlich nicht nur Aufrufe an unsere Library (CALL LINPUT, CALL TASTE), sondern auch an die von QuickBASIC.

Wie gesagt, QuickBASIC besitzt seine eigene Library. Und die enthält alle Routinen zur Ausführung der Anweisungen PRINT, WHILE, INPUT und so weiter. Jede dieser Anweisungen ruft eine Prozedur der QuickBASIC-Library auf, so wie CALL TASTE eine Prozedur unserer eigenen Library aufruft.

Das geladene QuickBASIC-System QB.EXE enthält alle Programme, die zur Ausführung unserer BASIC-Anweisungen benötigt werden (zusätzlich natürlich noch den Programm-Editor und den Compiler).

Im Speicher werden drei Programme benötigt, um CONVERT auszuführen:

- das Hauptprogramm CONVERT
- unsere User-Library
- die System-Library von QuickBASIC.

9.2 Linken mit BRUN20.LIB

QB.EXE enthält viel mehr als nur Programme zur Ausführung unserer BASIC-Anweisungen. Zusätzlich enthält diese Datei noch den Programm-Editor und den eigentlichen Compiler.

Die »reine« System-Library befindet sich allerdings auch auf der Quick-BASIC-Systemdiskette. Eigentlich gibt es zwei verschiedene Librarys, BRUN20.LIB und BRUN20.EXE.

Für uns wichtig ist momentan nur die Library BRUN20.LIB. Sie ist nur zusammen mit der Datei BRUN20.EXE brauchbar. BRUN20.EXE ist ein pures Maschinensprache-Programm, eigentlich mehr eine Programm-Sammlung. Dieses Programm ist ein »Laufzeitmodul«, das alle(!) Maschinenprogramme zur Ausführung der verschiedensten BASIC-Anweisungen enthält.

Die Library BRUN20.LIB ist im Grunde eine Art Tabelle. Die Einträge in dieser Tabelle sagen aus, wo sich innerhalb des Laufzeitmoduls BRUN20.EXE die einzelnen Programmteile zur Ausführung von PRINT, GOTO und so weiter befinden.

Und nun kommt der zweite Linker LINK.EXE ins Spiel. LINK erzeugt keine User-Librarys, dafür jedoch Stand-alone-Programme. Angenommen, wir kompilieren unser Hauptprogrammm CONVERT und lassen anschließend LINK auf die erzeugte Objektdatei CONVERT.OBJ los. LINK verbindet die Objektdatei mit der Library von QuickBASIC.

Die Frage ist, mit welcher Library? Ich sagte Ihnen ja, daß QuickBASIC über zwei Libraries verfügt, über BRUN20.LIB (zu der das Laufzeitmodul BRUN20.EXE gehört) und BCOM20.LIB. Welche Library benutzt wird, bestimmen wir beim Kompilieren. Beide Libraries können wir für unterschiedliche Zwecke einsetzen. Wenn wir angeben, daß LINK die Library BRUN20.LIB benutzen soll, passiert folgendes:

LINK analysiert CONVERT.OBJ und hält in seinen Tabellen fest, welche BASIC-Anweisungen unser Hauptprogramm enthält. Dann liest LINK die Library BRUN20.LIB ein. Sie enthält zu jedem Label eine »Adresse«. Die Adresse ist gewissermaßen die »Hausnummer« des betreffenden Maschinenprogramms im Laufzeitmodul BRUN20.EXE. Und diese Adresse setzt LINK in unser Hauptprogramm ein. Auf diese Weise ersetzt LINK überall in CONVERT Label durch die Adresse der zugehörigen Maschinenprogramme innerhalb von BRUN20.EXE. Und das Gleiche passiert mit den Aufrufen an unsere User-Library. LINK setzt für jeden Aufruf (zum Beispiel *CALL TASTE*) die Adresse des aufgerufenen Programms in USERLIB.EXE ein.

Das Resultat ist die lauffähige Datei CONVERT.EXE. Lauffähig ist dieses Programm jedoch nur unter gewissen Voraussetzungen. Es enthält nun Sprünge zu bestimmten Stellen im Laufzeitmodul BRUN20.EXE und in unserer User-Library USERLIB.EXE. Das heißt, CONVERT.EXE läuft nicht ohne diese beiden Dateien.

In der Praxis passiert folgendes: Sie rufen CONVERT.EXE auf. Dieses Programm lädt automatisch die benötigten Dateien BRUN20.EXE und USERLIB.EXE nach. Im Speicher befinden sich nun alle drei Maschinenprogramme. CONVERT ist nun tatsächlich lauffähig!

Probieren wir das Ganze aus. Zuerst müssen wir CONVERT jedoch ein letztes Mal kompilieren, um die Objektdatei CONVERT.OBJ zu erzeugen.

Verlassen Sie bitte QuickBASIC. Alles weitere spielt sich auf der Betriebssystem-Ebene ab. Diesmal kompilieren wir nicht mit unseren Batch-Dateien. Sie sind aus zwei Gründen ungeeignet:

1. DC.BAT und PC.BAT kompilieren mit den Optionen »/d« (Austesten) und »/q« (auf Geschwindigkeit optimieren). Diese Optionen behalten wir zwar bei. Zusätzlich benötigen wir jedoch für CONVERT die Option »/e«, um die Fehlerbehandlung einzuschalten (ebenso wie bei der Kompilierung von CONVERT im Speicher).

2. Wir müssen QuickBASIC beim Kompilieren mitteilen, daß CONVERT Aufrufe an die User-Library USERLIB.EXE enthält. Sonst wird sich QuickBASIC beim Kompilieren des Ausdrucks *Call Taste* fragen, wo denn bitteschön die Prozedur TASTE zu finden ist!

   Die Benutzung einer User-Library geben wir mit der Option »/l« an. Diese Option müssen Sie immer verwenden, wenn das zu kompilierende Programm eine User-Library benutzt! Sonst wird QuickBASIC beim Kompilieren Fehler melden, sobald es einen Aufruf an die User-Library entdeckt.

Hoffentlich wissen Sie noch, wie ohne unsere Batch-Files kompiliert wird. Der folgende Aufruf setzt als aktuelles Laufwerk (Verzeichnis) die Arbeitsdiskette (das Verzeichnis QB) voraus. Da wir ohne die Batch-Files arbeiten, ist der Aufruf für Disketten- und für Festplattenbesitzer identisch.

```
QB CONVERT /L /D /Q /E;
```

QuickBASIC erzeugt nun die Objektdatei CONVERT.OBJ. Ich sagte bereits, daß wir beim Kompilieren angeben müssen, welche Library beim Linken zu verwenden ist. Wir wollen mit BRUN20.LIB linken. Wie geben wir das an?

Sie haben es bereits angegeben! BRUN20.LIB ist als »Standard-Library« vorgegeben. Ohne spezielle Kompilieroptionen wird beim Kompilieren in der erzeugten Objektdatei automatisch vermerkt, daß später beim Linken BRUN20.LIB zu verwenden ist.

Übrigens: CONVERT befindet sich bereits in kompilierter Form im Verzeichnis OBJFILES (CONVERT.OBJ).

Nun kommt der Linkvorgang. Wie erläutert, benötigen wir nicht BUILDLIB.EXE (wir wollen keine User-Library erstellen), sondern LINK.EXE. Kopieren Sie LINK.EXE von der QuickBASIC-Diskette auf unsere Programmdiskette (Fesplatte: in das Verzeichnis SYSTEM20).

Da auch unsere User-Library benötigt wird, kopieren Sie USERLIB.EXE ebenfalls auf die Diskette (in das Verzeichnis SYSTEM20). Legen Sie nun diese Diskette ein (beziehungsweise gehen Sie in das Verzeichnis SYSTEM20).

Wenn wir LINK.EXE auf die erzeugte Objektdatei loslassen, erhalten wir eine .EXE-Datei, die unser Hauptprogramm CONVERT in ausführbarer Form enthält. Dieses Programm wird zusammen mit den Maschinensprache-Programmen USERLIB.EXE (unsere Routinen) und BRUN20.EXE (die QuickBASIC-Routinen) lauffähig sein.

Im aktuellen Verzeichnis befinden sich alle benötigten Programme, also können wir den Linker aufrufen. Auf das für uns Wichtige reduziert, lautet die Syntax beim Aufruf:

```
LINK Objektdatei,[ausführbare Datei];
```

- Objektdatei: zu linkende Datei mit der Erweiterung .OBJ
- ausführbare Datei: die erzeugte ausführbare Datei mit der Erweiterung .EXE

Die Angabe »ausführbare Datei« ist optional. Ohne diese Angabe besitzt die ausführbare Datei bis auf den Zusatz .EXE den gleichen Namen wie die Objektdatei.

Geben Sie also ein: `LINK CONVERT;`

Wie erläutert wird der Linker nun die Objektdatei CONVERT.OBJ analysieren und Tabellen erstellen, die alle Aufrufe an die beiden Routinensammlungen USERLIB.EXE und BRUN20.EXE enthalten. Ein Aufruf an USERLIB.EXE wäre zum Beispiel *Call Taste(..)* oder *Call PullDown(..)*. Ein Aufruf an BRUN20.EXE ist jede BASIC-Anweisung wie GOTO oder NEXT.

Aufrufe an Prozeduren in der User-Library (*Call Taste*) ersetzt LINK durch Sprünge zu den Adressen, an denen sich diese Programmteile innerhalb von USERLIB.EXE befinden.

Das Gleiche macht LINK mit den Aufrufen an die QuickBASIC-Library. Wo sich innerhalb von BRUN20.EXE ein bestimmtes Programm befindet, steht in den Tabellen der Library BRUN20.LIB. Zum Beispiel wird PRINT durch einen Sprung zu der Adresse der PRINT-Routine in BRUN20.EXE ersetzt.

Denken Sie daran: LINK benötigt die Library BRUN20.LIB. Sie merken das spätestens dann, wenn sich BRUN20.LIB nicht auf Ihrer Systemdiskette befindet.

Wenn der Linker BRUN20.LIB nicht findet, erhalten Sie folgende Fehlermeldung:

»Ich kann BRUN20.LIB nicht finden. Bitte neuen Pfad eingeben:«

Sie müssen den Linklauf nicht unbedingt abbrechen. Geben Sie einfach den kompletten Pfad inklusive Dateiname an.

Auf der Diskette/Platte befinden sich nun (außer LINK.EXE) drei ausführbare Dateien: USERLIB.EXE, BRUN20.EXE und CONVERT.EXE. Und mehr brauchen wir nicht!

Geben Sie bitte ein: `CONVERT`

CONVERT.EXE wird aufgerufen und die beiden anderen .EXE-Dateien nachgeladen. Alle benötigten Programme befinden sich im Speicher und CONVERT läuft!

Unser Gesamtprogramm ist etwa 110 Kbyte groß. Allerdings ist nicht alles »unser« Programm. Fairerweise sollte man zugestehen, daß der »größte Brocken«, BRUN20.EXE, von Microsoft stammt. Und genau dieses Programm dürfen Sie keinesfalls weitergeben, ohne rechtliche Schwierigkeiten zu riskieren!

Das heißt, daß alle »Eigenanwender« nun zufriedengestellt sind. Sie können Ihre Programme – wenn sie endgültig ausgetestet sind! – wie erläutert »behandeln« und anschließend ohne Kompilieren oder Linken jederzeit benutzen.

Wenn Sie Programme weitergeben wollen, ist das nächste Kapitel genau richtig für Sie. Es zeigt, wie Sie mit der alternativen Library BCOM20.LIB ein echtes Stand-alone-Programm erzeugen, das Sie jederzeit weitergeben dürfen. Denn das Laufzeitmodul BRUN20.EXE wird nicht mehr benötigt. Die erzeugte .EXE-Datei ist direkt lauffähig!

Doch zuvor fasse ich den Ablauf beim »Linken mit BRUN20« kurz zusammen. Eigentlich ist er recht simpel, aber ich wollte Ihnen auch die Hintergründe aufzeigen. Sonst kompilieren und linken Sie, ohne zu wissen, was eigentlich dabei passiert (und das Handbuch schweigt sich darüber aus).

1. Das Hauptprogramm wird kompiliert, um eine Objektdatei zu erzeugen. Vergessen Sie auf keinen Fall den Schalter »/l«, wenn das Hauptprogramm eine User-Library verwendet! Und achten Sie darauf, daß sich die User-Library im aktuellen Verzeichnis (oder im Zugriffspfad) befindet!

2. Die erzeugte Objektdatei wird mit LINK.EXE in eine ausführbare .EXE-Datei umgewandelt. Die Syntax (gegenüber dem Handbuch verkürzt):

   ```
   LINK Ojektdatei,[ausführbare Datei];
   ```

   Beispiel: `LINK CONVERT,TEST;`

   erzeugt aus CONVERT.OBJ die ausführbare Datei TEST.EXE. Beim Linken muß sich außer der User-Library auch die QuickBASIC-Library BRUN20.LIB im aktuellen Verzeichnis befinden!

3. Das erzeugte Programm wird mit dem Namen des Hauptprogramms aufgerufen. Da USERLIB.EXE und BRUN20.EXE nachgeladen werden, müssen sich beide Dateien im aktuellen Verzeichnis befinden!

9.3 Linken mit BCOM20.LIB

Die Erzeugung eines zusammen mit BRUN20.EXE lauffähigen Programms ist recht einfach. Das Hauptprogramm wird noch einmal kompiliert und der Linker aufgerufen. Mit etwas Übung erledigen Sie das Ganze in ein bis zwei Minuten.

Ein echtes Stand-alone-Programm zu erzeugen, ist erheblich aufwendiger. Jedes einzelne BASIC-Modul müssen Sie neu kompilieren!

»Stand-alone-Programm« heißt, am Ende der gesamten Prozedur soll sich ein(!) .EXE-File auf der Diskette/Platte befinden. In diesem Programm sind alle benötigten Teile integriert, das Hauptprogramm, unsere Library und die benötigten Teile von QuickBASIC.

LINK.EXE kann keineswegs nur eine Objektdatei in eine ausführbare Datei umwandeln. Der Linker ist in der Lage, beliebig viele Objektdateien zu einem .EXE-File zusammenzubinden!

Dabei gibt es jedoch ein Problem: Im letzten Abschnitt linkten wir mit der Library BRUN20.LIB. Das führt dazu, daß in unseren Programmen BASIC-Anweisungen einfach durch Sprünge zu den entsprechenden Routinen von BRUN20.EXE ersetzt werden.

Diesmal wollen wir jedoch ohne BRUN20.EXE auskommen. Also müssen die benötigten Routinen zur Ausführung von BASIC-Anweisungen in die erzeugte .EXE-Datei integriert werden.

Ob die benötigten Routinen integriert werden oder nicht, hängt von der verwendeten QuickBASIC-Library ab. Integriert werden sie, wenn wir beim Kompilieren angeben, daß mit der Library BCOM20.LIB gelinkt werden soll. Dazu kompilieren Sie mit der Option »/o«.

BCOM20.LIB ist nicht nur eine Tabelle, die Label und die zugehörigen Adressen im Programm BRUN20.EXE enthält. In BRUN20.LIB befinden sich außer den Label die kompletten Programme.

Der Linker wird sich beim Binden der Objektmodule aus der Library BCOM20.LIB alle Programme schnappen, die zur Ausführung unserer Anweisungen benötigt werden. Und diese Teile werden in die erzeugte .EXE-Datei integriert. Wir erhalten ein Programm, das alle benötigten Module enthält, um völlig allein lauffähig zu sein!

Da bereits beim Kompilieren mit der Option »/o« anzugeben ist, welche Library beim Linken verwendet wird, müssen wir jedes einzelne Modul unserer User-Library neu kompilieren.

Da in unseren Kompilier-Batches die Option »/o« fehlt, kompilieren wir wieder »von Hand«. Da ein Stand-alone-Programm erst nach langwieriger Entwicklungsarbeit einmalig erzeugt wird, lohnt es sich nicht, einen eigenen Batch zu schreiben. Kompilieren Sie der Reihe nach die benötigten Programme ROUTINEN.BAS, INPUT.BAS, PULLDOWN.BAS und CONVERT.BAS. Geben Sie außer »/d« und »/q« die Option »/o« an (Verwendung von BCOM20.LIB).

```
QB ROUTINEN /D/Q/O;
QB INPUT /D/Q/O;
QB PULLDOWN /D/Q/O;
QB CONVERT /D/Q/O/E;
```

Vergessen Sie auf keinen Fall, beim Kompilieren des Hauptprogramms mit »/e« die Option »On Error« zu aktivieren!

MASKE.BAS wird nicht kompiliert, da CONVERT die Eingabemaske nicht benötigt. Diese niemals aufgerufene Prozedur würde das erzeugte ausführbare Programm nur unnötig »aufblähen«.

Und sollten Sie sich wundern, daß beim Kompilieren von CONVERT diesmal nicht wie zuvor »/l« (Benutzung der User-Library) angegeben wird: Alle benötigten Module werden in die erzeugte Datei integriert. Nicht nur das Hauptprogramm und Teile der Library BCOM20.LIB, sondern auch die Objektmodule, die die User-Library bilden. Daher wird die USERLIB.EXE nicht mehr benötigt.

Die neu erzeugten Objektmodule befinden sich nun auf der Arbeitsdiskette (im Verzeichnis QB). Alle Module linken wir zu einem ausführbaren Programm. Außer den kompilierten BASIC-Programmen benötigen wir jedoch noch die Assembler-Routinen. Die zugehörigen Objektdateien befinden sich auf der Programmdiskette (im Verzeichnis SYSTEM20).

BCOM20.LIB ist eine extrem umfangreiche Library. Diskettenbesitzer sehen sich nun vor Probleme gestellt. Versuchen Sie bitte nicht, diese Datei noch auf unserer Programmdiskette unterzubringen.

Richten Sie sich eine eigene Diskette zum Linken mit BCOM20.LIB ein. Diese Diskette muß folgende Systemdateien enthalten:

- BCOM20.LIB, die QuickBASIC-Library
- LINK.EXE, das Linkprogramm
- und alle zu linkenden Objektdateien

Kopieren Sie bitte außer den beiden Systemdateien alle erzeugten Objektmodule von der Arbeitsdiskette auf diese Diskette. Kopieren Sie auch die Assembler-Objektdateien auf die Diskette. Sie enthält nun folgende Dateien:

- Die (mit der Option »/o« kompilierten) aus BASIC-Programmen erzeugten Objektdateien ROUTINEN.OBJ, INPUT.OBJ, PULLDOWN.OBJ und CONVERT.OBJ.
- Die aus Assembler-Programmen erzeugten Objektdateien PREFIX-.OBJ, INT86.OBJ, PRINTF.OBJ und WINDOWS.OBJ.
- Die QuickBASIC-Library BCOM20.LIB und den Linker LINK.EXE.

Auch als Besitzer einer Festplatte sollten Sie all diese Dateien in ein eigenes Verzeichnis kopieren. LINK.EXE und BRUN20.LIB besitzen Sie nun zwar doppelt, aber was macht das schon bei 20 oder mehr Megabyte. Würden Sie die erzeugten Objektdateien einfach in Ihr Verzeichnis SYSTEM20 kopieren, überschreiben Sie dabei die »alten« Objektdateien, mit denen die User-Library aufgebaut wurde. Und das heißt, Sie müssen all diese Dateien wieder per Batch-File aufbauen, wenn Sie die User-Library neu aufbauen wollen!

Dem Linker werden nun alle »zu linkenden« Objektdateien angegeben, und zwar in einer bestimmten Reihenfolge:

1. Als erste Objektdatei geben Sie das Hauptprogramm an (CONVERT.OBJ).
2. Als erste der aus Assembler-Programmen erzeugten Objektdateien geben Sie PREFIX.OBJ an.

Davon abgesehen, ist die Reihenfolge Ihnen überlassen.

```
LINK CONVERT ROUTINEN INPUT PREFIX INT86 PRINTF WINDOWS PULLDOWN,CONVERT;
```

LINK bindet alle angegebenen Objektdateien zur ausführbaren Datei CONVERT.EXE. Zur Ausführung dieses Programms wird keine weitere Datei mehr benötigt.

Nur die wirklich benötigten Teile aus der Library BCOM20.LIB werden in CONVERT.EXE eingebunden (was sollte auch die Einbindung der COLOR-Routine, wenn unser Programm keine COLOR-Anweisung enthält?).

Die erzeugte .EXE-Datei ist daher erheblich kleiner als die drei Programme, die beim Linken mit BRUN20.LIB benötigt wurden. Denn das Laufzeitmodul BRUN20.EXE enthält die komplette Library, egal ob tatsächlich alle BASIC-Befehle verwendet werden oder nicht.

Übrigens: Auf der Diskette zum Buch befindet sich CONVERT als Standalone-Programm im Stammverzeichnis (CONVERT.EXE).

Zusammenfassung:

1. LINK erzeugt aus Objektdateien ausführbare Programme. LINK benötigt zum Binden der Dateien eine QuickBASIC-Library, entweder BRUN20.LIB oder BCOM20.LIB.
2. Linken mit BRUN20.LIB:
   - Zur Ausführung wird das »Laufzeitmodul« BRUN20.EXE und – falls verwendet – die User-Library USERLIB.EXE benötigt.
   - Das Hauptprogramm wird kompiliert und eine Objektdatei erzeugt. Falls das Hauptprogramm eine User-Library verwendet, muß mit der Option »/l« kompiliert werden. CONVERT kompilieren Sie mit folgendem Aufruf (Achtung: »On Error« muß mit »/E« aktiviert sein):

     ```
     QB CONVERT /L /D /Q /E;
     ```

- Beim Aufruf von LINK wird nur das kompilierte Hauptprogramm angegeben. LINK greift auf die Library BRUN20.LIB zu (sollte sich im aktuellen Verzeichnis befinden). Zum Linken von CONVERT genügt der Aufruf:

  ```
  LINK CONVERT;
  ```

- Die erzeugte .EXE-Datei ist zusammen mit BRUN20.EXE (und USERLIB.EXE) lauffähig.

3. Linken mit BCOM20.LIB:

   - BCOM20.LIB ist eine alternative QuickBASIC-Library zur Erzeugung von Stand-alone-Programmen (allein lauffähigen Programmen).

   - Alle(!) Programm-Module werden neu kompiliert. Dabei muß die Option »/o« aktiviert sein, die die Verwendung von BCOM20.LIB beim Linken anzeigt. CONVERT erfordert folgende Kompilationen:

     ```
     QB ROUTINEN /D/Q/O;
     QB INPUT /D/Q/O;
     QB PULLDOWN /D/Q/O;
     QB CONVERT /D/Q/O/E;
     ```

   - LINK wird unter Angabe aller verwendeten Objektdateien aufgerufen. Zuerst geben Sie immer das Hauptprogramm an. Wenn auch Assembler-Module gelinkt werden, geben Sie vor allen anderen Assembler-Teilen die Datei PREFIX.OBJ an. LINK greift auf die Library BCOM20.LIB zu (aktuelles Verzeichnis). Am Beispiel CONVERT:

     ```
     LINK CONVERT ROUTINEN INPUT PREFIX INT86 PRINTF WINDOWS PULL-
     DOWN,CONVERT;
     ```

   - Die erzeugte .EXE-Datei ist ohne Einschränkungen lauffähig.

9.4 Die optimale Systemeinrichtung

Diskettenbesitzer sollten sich entweder schnellstens eine Festplatte zulegen oder ihr QuickBASIC-Systems so einrichten:

1. Arbeitsdiskette
   - QB.EXE
   - BASIC-Programme
2. Programmdiskette
   - BUILDLIB.EXE
   - BRUN20.LIB
   - BRUN20.EXE
   - LINK.EXE
   - DC.BAT
   - DL.BAT

   Auf diese Diskette kopieren Sie vor dem Linken mit BRUN20.LIB das kompilierte Hauptprogramm (z.B. CONVERT.OBJ) und USER-LIB.EXE (falls das Hauptprogramm die User-Library verwendet).
3. Diskette zum Kompilieren mit BCOM20.LIB:
   - BCOM20.LIB
   - LINK.EXE

   Bevor Sie linken, kopieren Sie alle zu bindenden Objektdateien auf diese Diskette.

Als Besitzer einer Festplatte richten Sie sich einfach drei Subdirectories entsprechend ein.

10

Strukturierte Programmierung mit der Version 3.0

Die Version 3.0 bietet einige neue Features. Dieses Kapitel handelt nicht von der Unterstützung des Arithmetik-Coprozessors oder EGA-Karte. Hier geht es um die zusätzlichen Möglichkeiten zur Schleifenbildung, um die Anweisungen DO..LOOP, SELECT..CASE, EXIT..DO und EXIT..FOR.

10.1 Konstanten

Zuvor eine Bemerkung zu der neuen Anweisung CONST. CONST erlaubt »echte« Konstanten. Ich verwendete bisher notgedrungen Variablen als Konstanten. Denken Sie an die Booleschen Variablen <True> und <False>. Dabei besteht immer die Gefahr, die Werte dieser Variablen versehentlich zu ändern. Es sind nun mal keine Konstanten.

Dank der Version 3.0 ist diese Notlösung nicht mehr nötig. CONST deklariert echte Konstanten, deren Werte wir nicht im Programmlauf ändern können.

```
CONST Name = Ausdruck[,Name = Ausdruck ...]
```

Konstanten sind übrigens immer global, also allen Prozeduren in dem Programm bekannt, in dem sie deklariert werden. Es spricht auch nichts dagegen. Denn im Gegensatz zu Variablen ist eine versehentliche Änderung des Wertes einer Konstanten ausgeschlossen.

In COMDEF.BAS deklarieren wir einige Variablen als global.

```
'       * Datentyp Boolean *
Common Shared False, True
```

In INIT.BAS weisen wir diesen Variablen Werte zu, die während des gesamten Programmlaufs (hoffentlich!) nicht verändert werden.

```
'        * Datentyp Boolean *
False      = Ø
True       = 1
```

CONST deklariert nicht nur Konstanten, sondern initialisiert sie gleichzeitig. Daher können wir unsere »Pseudokonstanten« nun erheblich einfacher deklarieren und initialisieren.

```
'        * Boolesche Wahrheitswerte *
CONST True = 1, False = Ø
```

Diese Anweisung in COMDEF.BAS ist ein gleichwertiger Ersatz für die vorige Deklaration und Initialisierung. Die Änderung muß(!) in COMDEF.BAS stattfinden. Konstanten sind zwar automatisch global, aber natürlich nur für das Programm, in dem die Anweisung CONST steht. Und von den beiden Booleschen Werten TRUE und FALSE sollen ja all unsere Programme wissen!

Ach übrigens: Konstanten müssen vor der ersten Benutzung deklariert sein. Also bitte nicht so:

```
Print 2 * MWSt
CONST MWSt = 14
```

10.2 Schleifenstrukturen

3.0 stellt Ihnen eine weitere Schleifenstruktur zur Verfügung, DO..LOOP. Diese Struktur ist allerdings enorm vielseitig. Kriterien für Schleifenstrukturen sind üblicherweise:

- wo wird getestet: am Schleifenanfang oder am Schleifenende?
- wann wird die Schleife verlassen: wenn die angegebene Bedingung erfüllt oder nicht mehr erfüllt ist?

DO..LOOP erlaubt alle Kombinationsmöglichkeiten! Außer WHILE gibt es den zusätzlichen »Testbefehl« UNTIL. Mit WHILE wird die Schleife wie üblich verlassen, wenn die Bedingung nicht mehr erfüllt ist (»Wenn Bedingung erfüllt, führe aus«). UNTIL ist die Umkehrung von WHILE (»Wenn Bedingung erfüllt, führe nicht mehr aus«).

Beide Testbefehle dürfen sich am Schleifenanfang oder -ende befinden.

1. Testen am Schleifenanfang

1a) Test mit WHILE	1b) Test mit UNTIL

`i = 1`
`Do While i <> 2`
`  Print i`
`  i = i + 1`
`Loop`
	`i = 1`
	`Do Until = 2`
	`  Print i`
	`  i = i + 1`
	`Loop`

2. Testen am Schleifenanfang

2a) Test mit WHILE	2b) Test mit UNTIL

`i = 1`
`Do`
`  Print i`
`  i = i + 1`
`Loop While i <> 2`
	`i = 1`
	`Do`
	`  Print i`
	`  i = i + 1`
	`Loop Until i = 2`

Alle vier Varianten geben genau einmal den Inhalt der Variablen <i> aus, also eine Eins. Ob Sie als Testbefehl WHILE oder UNTIL verwenden, ist reine Geschmackssache. Ich würde sagen, es hängt davon ab, wir Ihr Gehirn funktioniert.

Wenn es so wie mein eigenes funktioniert, überlegen Sie sich bei der Schleifenkonstruktion, wann die Schleife verlassen werden soll und nicht, wann sie wiederholt werden soll. Und dann sind Sie ohne UNTIL gezwungen, Ihre »Gehirnwindungen zu verbiegen«.

Sie haben bei der Konstruktion einer Schleife folgende Vorstellung: »Die Schleife soll durchlaufen werden, bis eine bestimmte Bedingung eintritt«. Bisher mußten Sie diese Bedingung passend für WHILE umformulieren: »Die Schleife soll durchlaufen werden, wenn die gegenteilige Bedingung nicht erfüllt ist«. Eine verzwickte Angelegenheit, wenn es sich nicht nur um eine, sondern um mehrere mit AND und OR verknüpfte Bedingungen handelt:

Formulierung 1

»Durchlaufen, bis <i> gleich der Länge von <S$> ist oder das Zeichen in <S$> an der Position <i> gleich »!« oder gleich »*« ist«

Mit UNTIL können Sie ohne verzwickte Überlegungen direkt die Originalbedingung hinschreiben:

```
Until i=Len(S$) Or Mid$(S$, i, 1)="!" Or Mid$(S$, i, 1)="*"
```

Ohne UNTIL sind zuvor alle Teilbedingungen und die Verknüpfungen zwischen den Teilbedingungen logisch zu negieren:

Formulierung 2 (Negation der Teilbedingungen und der Verknüpfungsoperatoren)

»Durchlaufen, solange <i> ungleich der Länge von <S$> ist und das Zeichen in <S$> an der Position <i> ungleich »!« und ungleich »*« ist«

Und nun können Sie die Schleife endlich konstruieren:

```
While i<>Len(S$) And Mid$(S$, i, 1)<>"!" And Mid$(S$, i, 1)<>"*"
```

10.3 Vorzeitiges Verlassen von Schleifen

Keine Geschmackssache ist es allerdings, ob die Bedingung am Anfang oder am Ende der Schleife getestet wird. Das hängt allein von der jeweiligen Aufgabenstellung ab. Am Anfang müssen Sie immer dann testen, wenn die Schleife vielleicht nicht ein einziges mal durchlaufen werden soll.

Ein gutes Beispiel dafür ist unsere Routine COMPRESS, die überflüssige Leerzeichen am Stringende entfernt.

Mit EXIT DO kann jede Form der DO-Schleife vorzeitig verlassen werden. Mit dieser Erweiterung können wir COMPRESS nun eleganter als zuvor formulieren.

```
Sub Compress(S$) Static
  i = Len(S$)
  Do
    If Mid$(S$, i, 1) <> " " Then Exit Do
    i = i - 1
  Loop Until i = 1
  S$ = Left$(S$, i)
End Sub
```

Die Schleife tastet sich vom Stringende in Richtung Stringanfang. Wenn das aktuelle Zeichen kein Leerzeichen ist, wird sie mit EXIT DO verlassen. Ansonsten wird sie wiederholt, bis der Stringanfang erreicht ist.

Diese Methode funktioniert fast immer. Leider nur fast immer, denn es gibt genau eine Ausnahme, und zwar den Aufruf

```
S$ = "":  Call Compress(S$)
```

<S$> ist leer. <i> wird mit der Länge von <S$> initialisiert, also mit 0. Da der Test der Bedingung am Schleifenende stattfindet, wird die Schleife auf alle Fälle zumindest einmal ausgeführt.

Mid$(S$, i, 1) übergibt das Zeichen Nummer <i>, also das Zeichen Nummer 0. Und daß QuickBASIC mit diesem Zeichen so wenig anzufangen weiß wie wir, erhalten wir eine Fehlermeldung.

Um selbst für diesen »blödsinnigen« Aufruf vorzusorgen, prüfen wir einfach am Schleifenanfang.

```
Sub Compress(S$) Static
  i = Len(S$)
  Do Until i = Ø
    If Mid$(S$, i, 1) <> " " Then Exit Do
    i = i - 1
  Loop
  S$ = Left$(S$, i)
End Sub
```

Wenn der String leer ist, wird die Schleife nicht einmal durchlaufen. Ob Sie mit UNTIL oder mit WHILE testen, ist wieder Geschmackssache. Mit WHILE: *Do While i <> 0.*

An unserer Routine TASTE will ich Ihnen noch einmal zeigen, welche Umständlichkeiten uns die DO-Schleife erspart. TASTE war so aufgebaut:

```
Sub Taste.....
  Key$ = ""
  While Key$ = ""
  .....
  .....
  Wend
End Sub
```

Die Schleife muß auf alle Fälle einmal durchlaufen werden. Und da wir uns beim Testen am Schleifenanfang dessen nie sicher sein können, müssen wir <Key$> in solchen Fällen immer passend zur Schleifenbedingung initialisieren. Denn wir können nie ausschließen, daß <Key$> bereits beim Aufruf ein Zeichen enthält. Und diese »Extra-Initialisierung« finden Sie in diesem Buch vor beinahe jeder WHILE-Schleife.

Mit DO testen wir einfach am Schleifenende:

```
Sub Taste(Key$, Escape) Static
  Escape = False
  Do
    Key$ = Inkey$
  Loop While Key$ = ""
  If Len(Key$) > 1 Then_
    Escape = True:_
    Key$ = Right$(Key$, 1)
End Sub
```

Die Schleife wird nun auf alle Fälle mindestens einmal durchlaufen und die Tastatur mit INKEY$ abgefragt. Diese Prozedur steht stellvertretend für eine Unmenge ähnlicher Fälle.

EXIT DO habe ich »so nebenbei« erwähnt. Diese Anweisung bereitet Ihnen sicher keine Probleme. Sie ist nicht unbedingt notwendig. Wir können sie in DO-Schleifen auch durch komplexere Schleifenbedingungen ersetzen.

```
Sub Compress(S$) Static
  i = Len(S$)
  Do Until i = Ø Or Mid$(S$, i, 1) <> " "
    i = i - 1
  Loop
  S$ = Left$(S$, i)
End Sub
```

Viel wichtiger ist EXIT FOR. Eine tolle Sache. Angenommen, Sie suchen im Array <Name$(1> bis <Name$(100)> den Namen »Maier«. Bisher war es sehr umständlich, sich den Index des betreffenden Arrayelementes zu merken. Drei Möglichkeiten standen zur Auswahl:

1. Index merken, Schleife weiterlaufen lassen

```
For i=1 to 1ØØ
  If Name$(i) = "Maier" Then Index = i
Next i
```

Die gefährlichste Methode. Wenn »Maier« mehrmals vorkommt, erhalten Sie den Index des letzten Elementes »Maier«, nicht des ersten! In der Praxis sollten Sie die Schleife vorzeitig abbrechen.

2. Index merken, Schleife mit GOTO verlassen

```
For i=1 to 1ØØ
  If Name$(i) = "Maier" Then Index = i: Goto Ausgang
Next i
Ausgang:
```

Dies ist sicher die unfeinste Methode. Wir programmieren in QuickBASIC, was soll da eine GOTO-Anweisung (ich hoffe, Sie erinnern sich nicht mehr an CONVERT – sonst wird's peinlich für mich)!

3. Index merken, Schleifenvariable manipulieren

```
For i=1 to 100
  If Name$(i) = "Maier" Then Index = i: i = 100
Next i
```

Die eleganteste Methode. Das »elegant« sollten Sie allerdings in Anführungszeichen setzen. So toll ist die Methode auch wieder nicht. Vor allem ist sie unverständlicher als die zweite Version. Das verpönte GOTO hat einen Vorteil: Man sieht sofort, daß die Schleife verlassen wird.

Nach diesem kleinen Ausflug unter dem Motto »Tausend Wege führen aus der Schleife«, kommen wir nun zum Weg Nummer 1001, der jedes Programmiererherz erfreut.

```
For i=1 to 100
  If Name$(i) = "Maier" Then Index = i: Exit For
Next i
```

Toll, finden Sie nicht auch? Man sieht sofort, was passiert, auch ohne das gehaßte GOTO.

10.4 Auswahl unter alternativen Bedingungen

SELECT CASE erspart uns immer wiederkehrende IF-Sequenzen. Erinnern Sie sich an UPLOWCASE?

```
.....
.....
If Char$ = "Ä" Then Char$ = "ä"
If Char$ = "Ö" Then Char$ = "ö"
If Char$ = "Ü" Then Char$ = "ü"
.....
.....
```

SELECT CASE wählt unter verschiedenen Möglichkeiten höchst effektiv die zutreffende aus.

```
Select Case Char$
  Case "Ä"
    Char$ = "ä"
  Case "Ö"
    Char$ = "ö"
  Case "Ü"
    Char$ = "ü"
End Select
```

Nach SELECT CASE wird ein Ausdruck angegeben, der mit den Konstanten in den folgenden CASE-Anweisungen verglichen wird. Trifft eine der Auswahlbedingungen zu, wird der folgende Anweisungsblock ausgeführt (also nicht unbedingt nur eine Anweisung wie in diesem Beispiel). Achtung: Der Anweisungsblock darf sich nicht in der gleichen Zeile wie die Selektion mit CASE befinden!

Falsch:
```
Select Case Char$
  Case "Ä" Char$ = "ä"
  Case "Ö" Char$ = "ö"
  Case "Ü" Char$ = "ü"
End Select
```

Ich gebe zu, in diesem Beispiel bringt SELECT CASE nicht allzuviel. Aber die Konstantenliste ist zum Glück sehr vielseitig. Sie können mit dem Wort TO Anfang und Ende eines zu testenden Bereichs angeben. Das Wort IF erlaubt zusätzlich die Verwendung von Vergleichsoperatoren zur Angabe der Bereiche.

```
Input "Name";N$
Select Case N$
  Case "A" to "F"
    Print "Alphabetisch im Bereich A bis F einzuordnen"
  Case Is > "F"
    Print "Kommt alphabetisch nach F"
End Select
```

Ich denke, dieses Beispiel spricht für sich. SELECT CASE ist so vielseitig, daß Ihnen bestimmt Unmengen an Anwendungen einfallen werden.

Nach CASE dürfen ganze Konstantenlisten folgen, in denen Sie alle besprochenen Möglichkeiten kombinieren können.

```
Input "Name";N$
Select Case N$
  Case "A" to "F", "Maier", Is > "X"
    Print "Beispiel für komplexe Konstantenliste"
End Select
```

Mit IF sieht das Ganze nicht nur umständlicher aus, sondern ist vor allem schwerer lesbar:

```
Input "Name";N$
If N$ >= "A" Or N$ <= "F" Or N$ = "Maier" Or N$ > "X" Then_
  Print "Beispiel für komplexe Konstantenliste"
```

Das war's, was die Version 3.0 in bezug auf strukturierte Programmierung zusätzlich bietet. Es sind nicht allzu viele Anweisungen, aber »genau die richtigen«. Vor allem die Deklaration echter Konstanten, die sehr vielseitige

DO-Struktur und die Möglichkeiten zum vorzeitigen Verlassen von Schleifen sind sehr erfreulich.

Wenn ich mir zum »BASIC-Teil« des Buches noch ein Abschlußwort erlauben darf: Für die Version 4.0 wünsche ich mir Rekursion, Pointer, Records, die uneingeschränkte Nutzung des Rechnerspeichers für Daten beliebiger Art (nicht nur große numerische(!) Arrays) und die Option, eigene Datentypen zu definieren.

Dann enthält QuickBASIC so ziemlich alles, was auch die Vorbilder Pascal und C enthalten – und ist dem Vorbild dank der User-Librarys sogar überlegen!

Ganz schön anspruchsvoll, meinen Sie? Sicher, aber schauen Sie sich einmal an, welche Entwicklung BASIC in letzter Zeit durchmacht. Es ist fast schon lächerlich, QuickBASIC mit den BASICs von Heimcomputern oder auch mit GW-BASIC vergleichen zu wollen. Es war ein langer Weg bis zu QuickBASIC, aber warum sollte die Entwicklung von BASIC nun enden?

11

QuickBASIC und Assembler

Mit QuickBASIC erstellte Programme sind schnell, sogar sehr schnell. Aber die höhere Programmiersprache, die Assembler völlig überflüssig macht, muß erst noch erfunden werden.

Das übliche Problem auf PCs ist die langsame Bildschirmausgabe. Schuld daran sind die BIOS-Routinen zur Zeichenausgabe, die extrem allgemein gehalten sind. Bei der Zeichenausgabe testen diese Routinen alles mögliche (Typ des Bildschirmadapters, aktive Bildschirmseite etc.) und bremsen dadurch die Ausgabegeschwindigkeit.

Üblicherweise benötigt man daher in jeder höheren Programmiersprache eine Assembler-Routine wie PRINTF, die die BIOS-Routinen umgeht und direkt in den Bildschirmspeicher schreibt.

Außer PRINTF stellte ich WINDOWS vor. Diese Routine greift ebenfalls direkt auf den Bildschirmspeicher zu und kopiert einen beliebigen rechteckigen Ausschnitt in ein BASIC-Array beziehungsweise schreibt den Inhalt dieses Arrays auf den Bildschirm zurück.

Diese beiden Routinen werde ich benutzen, um Ihnen zu zeigen, was zu beachten ist, wenn Sie Ihre QuickBASIC-Programme mit Assembler-Routinen unterstützen wollen.

Assembler und eine höhere Programmiersprache, das heißt immer, daß Sie die Schnittstellen des Compilers kennen müssen. Sie müssen wissen, wie von Assembler aus auf BASIC-Variablen zugegriffen wird, wie Sie diese Variablen lesen oder ihren Inhalt manipulieren können.

11.1 Assembler-Programme einbinden

Wenn Sie Erfahrungen mit BASIC-Interpretern (GW-BASIC) besitzen, kennen Sie ein weiteres Problem: Wie integriert man die Assembler-Routine in das BASIC-Programm. Aufgerufen wird ein Assembler-Programm mit

CALL unter Angabe der Adresse. Aber wie bekommen Sie das Programm gezielt in einen bestimmten Speicherbereich?

Bei Interpretersprachen haben sich mehrere »Standardtricks« durchgesetzt:

- Einen Speicherbereich schützen und das Programm mit BLOAD (binary load) in genau diesen Bereich laden.
- Das Assembler-Programm in Form von Data-Zeilen einbinden, die in einen geschützten Bereich gepokt werden.
- Das Programm Byte für Byte in ein BASIC-Array einlesen und vor dem Aufruf der Routine die Adresse des Arrays mit VARPTR ermitteln.

Bitte vergessen Sie all diese Tricks. Compiler wie QuickBASIC erlauben eine echte Einbindung, die nicht »durch die Hintertür« erfolgt.

Voraussetzung ist allerdings, daß Sie die Routine in die User-Library einbinden oder Module zu einem Stand-alone-Programm zusammenlinken.

In beiden Fällen bildet eine oder mehrere Objektdateien den Ausgangspunkt. Diese Objektdateien bearbeiten Sie mit BUILDLIB oder mit LINK. Eine von QuickBASIC erzeugte Objektdatei ist genauso aufgebaut wie die von einem Assembler wie MASM.

Das heißt, QuickBASIC ist es egal, welche Objektmodule Sie zu einer .EXE-Datei linken, ob es ein assemblierter Assembler-Quelltext oder ein kompilierter BASIC-Programmtext ist. In beiden Fällen enthält die erzeugte Objektdatei den Maschinencode und die zugehörigen Label.

Beim Linken der Module passiert das übliche. Der Linker (BUILDLIB beziehungsweise LINK) analysiert die Objektdateien und ermittelt, an welchen Adressen sich welche Prozeduren befinden. Er erzeugt ein .EXE-File, das den Maschinencode aller Objektdateien enthält, in dem jedoch bei allen Sprungbefehlen Label durch die zugehörigen Adressen ersetzt sind.

Um die Angabe der Adresse beim Aufruf mit CALL müssen Sie sich im Gegensatz zu GW-BASIC überhaupt nicht kümmern. Angenommen, Sie erstellen eine User-Library, die unter anderem die Assembler-Routine PRINTF enthält. Nach dem Linken mit BUILDLIB ist bekannt, an welcher Adresse sich PRINTF befindet. Nun schreiben Sie ein BASIC-Programm, das Aufrufe an die User-Library enthält. QuickBASIC kennt die Adressen aller Prozeduren in der User-Library, auch die von PRINTF. Beim Kompilieren Ihres Programms wird diese Adresse in das erzeugte Kompilat eingesetzt.

Allgemein ausgedrückt: Assemblierte Assembler-Programme behandelt QuickBASIC nicht im geringsten anders als kompilierte BASIC-Programme.

Die Prozeduren werden immer auf die gleiche Weise aufgerufen, CALL und Angabe der Parameterliste.

Theoretisch können Sie also Programme erstellen, deren Objektmodule aus den unterschiedlichsten Quellen stammen, aus kompilierten C-Programmen, Pascal-Programmen, Assembler-Programmen und so weiter.

In der Praxis scheitert diese Flexibilität jedoch an der unterschiedlichen Parameterübergabe. Zum Beispiel übergibt Turbo-Pascal einer aufgerufenen Prozedur Parameter auf völlig andere Art und Weise als QuickBASIC. QuickBASIC übergibt immer Adressen und niemals den Wert selbst, im Gegensatz zu Turbo-Pascal.

Das heißt, die aus einem QuickBASIC-Programm aufgerufene Pascal-Prozedur wird die übergebenen Parameter falsch interpretieren und umgekehrt.

Mit Assembler-Routinen gibt es keine derartigen Schwierigkeiten. Sie selbst entscheiden darüber, wie Ihr Assembler-Programm übergebene Parameter interpretiert, ob als Wert oder aber als Adresse, also als Pointer auf einen Wert.

Aber bevor wir uns um die Schnittstellen kümmern, zeige ich Ihnen, wie eine Assembler-Routine in der Praxis eingebunden wird. Die Diskette zum Buch enthält im Verzeichnis ASEMBLER die Datei DEMO.ASM, einen Assembler-Quelltext.

```
;DEMO: 'A' an der aktuellen Cursorposition ausgeben
;-------------------------------------------------
WorkSpace     equ 2Ø              ;Platz für lokale Variablen

Data          Segment Word Public 'Data'
Data          Ends
DGroup        Group Data

Code          Segment Byte Public 'Code'
              Assume CS:Code, DS:DGroup

              Public Demo
Demo          Proc Far

; ----- Vorbereitungen -----
              Push BP             ;BP retten und SP nach BP bringen
              Mov  BP,SP          ;subtrahieren von 'WorkSpace'
              Sub  SP, WorkSpace  ;=> Platz für lokale Variablen

; ----- Hauptprogramm -----
              Mov  AH,6           ;mit Funktion 6 von Interrupt 21H
              Mov  DL,65          ;Zeichen mit dem ASCII-Code 65
              Int  21H            ;ausgeben ('A')
```

```
; ----- Stack korrigieren/Register wiederherstellen -----
EndLoop:      Mov  SP, BP          ;Platzreservierung aufheben
              Pop  BP              ;BP wiederherstellen
              Ret

Demo          Endp
Code          Ends
              End
```

Das Mini-Hauptprogramm ist eigentlich relativ uninteressant. Es benutzt einen DOS-Interrupt, um das Zeichen »A« an der aktuellen Cursorposition auszugeben.

Viel wichtiger ist der »Rahmen« (Assembler-Anweisungen und Stackbehandlung). Um Schwierigkeiten zu vermeiden, sollten Sie die Assembler-Anweisungen in Ihre eigenen Programme übernehmen (die Deklarationen der Segmente etc.).

Für kleinere Assembler-Routinen ist der Stack der bei weitem wichtigste Datenbereich. Auf dem Stack übergibt QuickBASIC Parameter und auf ihm legen Sie auch die von Ihrem Programm verwendeten Daten ab. Der Zugriff erfolgt über den BasePointer BP. Um den Originalinhalt von BP nicht zu verlieren, wird BP auf den Stack gebracht (Push BP), bevor dieses Register mit dem aktuellen Wert des StackPointers SP initialisiert wird (Mov BP,SP). Anschließend wird <WorkSpace> subtrahiert, um Platz für eigene Variablen zu gewinnen.

Assemblieren Sie das Programm wie gewohnt. Kopieren Sie es auf die Programmdiskette, die ja unsere Objektdateien enthält. Rufen Sie nun BUILDLIB über unsere Batch-Datei auf, um die erzeugte Objektdatei in eine User-Library einzubinden.

Diskette: `DL DEMO` Platte: `PL DEMO`

Laden Sie QuickBASIC mit dem Parameter »/l«. Und nun schreiben Sie ein Hauptprogramm, das die Prozedur DEMO zehnmal aufruft.

```
For i=1 to 10
  Call Demo
Next i
```

Programmlauf: `AAAAAAAAAA`

Das war's auch schon. Das Herumärgern mit Segmenten und Offsets beim Aufruf einer in GW-BASIC eingebundenen Assembler-Routine können Sie also vergessen. Sie rufen Ihre Assembler-Routinen genauso wie Ihre BASIC-Routinen auf. Eingebunden werden die Routinen am bequemsten beim Linken, entweder mit BUILDLIB (User-Library) oder nach beendeter Programmentwicklung mit LINK (Stand-alone-Programm).

11.2 Stackverwaltung

Wenn man berücksichtigt, daß QuickBASIC die Rücksprungadresse (Segment/Offset) auf den Stack bringt, sieht der Stack nach der im Demoprogramm vorgenommenen Initialisierung so aus:

[BP+4]	Rücksprungadresse: Segment	Rücksprungadresse
[BP+2]	Rücksprungadresse: Offset	
BP =>	Gepushter Inhalt von BP	Nach 'Push BP'
[BP-2]	Temporäre Variable Ø	Eigene in der Assembler-Routine verwendete Variablen (je 2 Byte = 1 Wort)
[BP-4]	Temporäre Variable 1	
	.....	
	.....	
SP =>	Temporäre Variable 1Ø	

Dieses Schema enthält bereits eine Platzreservierung für in der Assembler-Routine verwendete Daten. Für das Demoprogramm ist diese Reservierung zwar nicht nötig. Sie werden jedoch kaum in der Lage sein, eine größere Routine zu schreiben, ohne einen bestimmten Speicherbereich für eigene Daten zu benutzen. Und genau dafür werden wir den Stack benutzen!

Zur Interpretation: Ganz »oben« auf dem Stack befindet sich die Rücksprungadresse von QuickBASIC.

Dann kommt der von uns auf den Stack gerettete Inhalt von BP (Push BP). Da der aktuelle Inhalt von SP nach BP gebracht wird (Mov BP,SP), weist BP im gesamten Programmlauf auf diese Adresse. Daran müssen Sie sich orientieren, wenn Sie Variablen auf dem Stack relativ zu BP adressieren!

Durch das Subtrahieren von 20 vom aktuellen Wert von SP wird der Stack-Pointer »nach unten versetzt«. Folgende Push-Aktionen finden 20 Byte unterhalb des aktuellen Wertes von BP statt. Der Bereich [BP-2] bis [BP-20] steht Ihnen nun für die von der Assembler-Routine benutzten Daten zur Verfügung. Sie können zum Beispiel 20 Wort- oder 40 Byte-Variablen in diesem Bereich unterbringen.

Zur Übung legen wir nun die ASCII-Codes der Zeichen »A« und »B« auf dem Stack im reservierten Bereich [BP-2] bis [BP-20] ab, laden diese Werte wieder und geben anschließend die zugehörigen Zeichen aus.

```
;DEMO1: 'A' und 'B' an der aktuellen Cursorposition ausgeben
;-----------------------------------------------------------
WorkSpace   equ 20              ;Platz für lokale Variablen
CharA       equ [BP-2]          ;Speicherplatz für A
CharB       equ [BP-4]          ;Speicherplatz für B

Data        Segment Word Public 'Data'
Data        Ends
DGroup      Group Data

Code        Segment Byte Public 'Code'
            Assume CS:Code, DS:DGroup

            Public Demo1
Demo1       Proc Far

; ----- Vorbereitungen -----
            Push BP             ;BP retten und SP nach BP bringen
            Mov  BP,SP          ;subtrahieren von 'WorkSpace'
            Sub  SP, WorkSpace  ;Platz für lokale Variablen

; ----- Hauptprogramm -----
            Mov  AX,65
            Mov  CharA,AX
            Mov  AX,66
            Mov  CharB,AX

            Mov  AH,6           ;mit Funktion 6 von Interrupt 21H
            Mov  DL,CharA       ;Zeichen mit dem ASCII-Code 65
            Int  21H            ;ausgeben ('A')

            Mov  AH,6           ;mit Funktion 6 von Interrupt 21H
            Mov  DL,CharB       ;Zeichen mit dem ASCII-Code 66
            Int  21H            ;ausgeben ('B')

; ----- Stack korrigieren/Register wiederherstellen -----
EndLoop:    Mov  SP, BP         ;Platzreservierung aufheben
            Pop  BP             ;BP wiederherstellen
            Ret

Demo1       Endp
Code        Ends
            End
```

Das Programm finden Sie unter dem Namen DEMO1.ASM. Assemblieren Sie es und erstellen Sie aus dem Objektfile eine User-Library (DL DEMO1 beziehungsweise PL DEMO1). Nach dem Aufruf mit Call Demo1 gibt die Routine den String »AB« aus.
Die Codes 65 und 66 werden in zwei am Anfang deklarierten Speicherplätzen (mit Wortbreite) gespeichert.

```
CharA       equ [BP-2]          ;Speicherplatz für A
CharB       equ [BP-4]          ;Speicherplatz für B
```

BP zeigt auf den gepushten Stackpointer. Darunter beginnt der reservierte Bereich. Relativ zu BP wird der erste Speicherplatz daher mit [BP-2] und der zweite mit [BP-4] adressiert. Jedenfalls dann, wenn wir unsere Daten immer in Wortbreite speichern (Verschwendung, solange nur ASCII-Codes zu speichern sind).

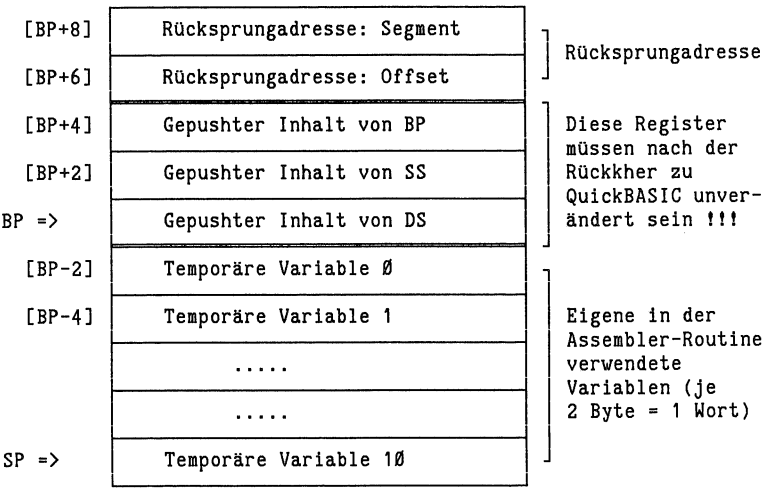
Eine Kleinigkeit fehlt noch. Bei der Rückkehr zu QuickBASIC müssen die Register BP, SS und DS den gleichen Inhalt wie beim Aufruf besitzen. Diese Register werden in beiden Demoprogrammen nicht beeinflußt. Allerdings nur, weil die Demoprogramme wirklich alles andere als umfangreich sind. In der Praxis müssen Sie außer BP auch SS und DS auf den Stack bringen und zum Schluß wiederherstellen. Nun kann ich Ihnen endlich den endgültigen »Programmrahmen« vorstellen.

11.3 Standardrahmen für Assembler-Routinen

```
WorkSpace    equ 20              ;lokale Variablen auf Stack

Data         Segment Word Public 'Data'
Data         Ends
DGroup       Group Data

Code         Segment Byte Public 'Code'
             Assume CS:Code, DS:DGroup

             Public ProgName
ProgName     Proc Far

; ----- Vorbereitungen -----
             Push BP             ;BP, SS und DS
             Push SS             ;müssen vor der Rückkehr
             Push DS             ;wiederhergestellt werden !!!
             Mov  BP,SP          ;BP = SP = 1.Parameter - 12
             Sub  SP,WorkSpace   ;Platz für lokale Variablen


; ----- Hauptprogramm
.....
.....
.....
```

```
; ----- Stack korrigieren/Register wiederherstellen -----
            Mov  SP, BP          ;Platzreservierung aufheben
            Pop  DS              ;Von QuickBASIC benötigte
            Pop  SS              ;Register wiederherstellen
            Pop  BP
            Ret

ProgName    Endp
Code        Ends
            End
```

Durch die beiden zusätzlichen Push-Befehle ändert sich der Stack-Zustand ein wenig.



Zum Glück ändert sich nichts an der Adressierung unserer Variablen. Die erste Variable adressieren Sie weiterhin mit [BP-2]. BP zeigt auf den letzten gespushten Wert DS. Darunter beginnt der 20 Byte große Freiraum bis zum Stackpointer, den wir für unsere internen Variablen benutzen.

11.4 Parameterübergabe

Alle Parameter an eine aufgerufene Prozedur übergibt QuickBASIC auf dem Stack. Andere Compiler übergeben meist sowohl Werte als auch Adressen. Zum Beispiel übergibt Turbo-Pascal bei der Übergabe nach Wert die Werte auf dem Stack, Zahlen oder komplette Strings. Bei der Übergabe nach Referenz wird natürlich die Adresse der Variablen auf den Stack gelegt.

QuickBASIC übergibt auf dem Stack immer Adressen, egal, ob Sie Parameter nach Wert oder nach Referenz übergeben!

Die übergebene Adresse ist ein Offset zum Datensegment von QuickBASIC. Wenn QuickBASIC eine Assembler-Routine aufruft, bleibt das aktuelle Datensegment unverändert. Das heißt, Sie benötigen nur diesen Offset, um auf einen übergebenen Parameter zuzugreifen.

Nun fragen Sie sich bestimmt, wie das bei dynamischen Arrays funktioniert, die sich ja außerhalb(!) des Datensegments befinden und – zumindest numerische Arrays – den gesamten Speicher nutzen können.

Es funktioniert so: Angenommen, Sie übergeben ein Element eines dynamischen Arrays. Das Element befindet sich außerhalb DS, QuickBASIC kann eigentlich gar keinen sinnvollen Offset zu DS übergeben. Aber nun kommt der Trick: Die Variable wird in einen freien Speicherbereich im Datensegment kopiert, in eine »temporäre Adresse«. Der verwendete Bereich wird nach Rückkehr aus der Prozedur wieder freigegeben. Und auf dem Stack übergibt uns QuickBASIC den Offset dieser temporären Adresse!

Wir können mit dem übergebenen Offset zu DS auf diese »Kopie« zugreifen, obwohl wir an das Original (das sich außerhalb DS befindet) nicht herankommen.

Das Gleiche passiert, wenn Variablen nach Wert übergeben werden oder wenn wir Ausdrücke übergeben. Der Aufruf *Call Test(3, A + 2)* übergibt zwei Ausdrücke.

Die Ergebnisse der beiden Ausdrücke werden in DS in temporären Adressen untergebracht und wir erhalten den zugehörigen Offset zu DS.

Für uns vereinfacht sich dadurch einiges. Egal, ob einfache Variablen übergeben werden, Ausdrücke oder Datenfeldelemente – immer befindet sich der betreffende Parameter in DS und wir können über den übergebenen Offset darauf zugreifen. Und ob wir nun auf die »echte« Variable oder nur auf eine temporäre Kopie zugreifen, kann uns zumindest bei »nur lesenden« Zugriffen egal sein.

Anders verhält es sich beim Schreiben, wenn wir also den Inhalt von Variablen manipulieren wollen. Manipulieren können wir nur Variablen, die per Referenz übergeben werden und deren »echte« Adresse wir erhalten. Sonst verändern wir nur den Inhalt der temporären Kopie, die nach Rückkehr aus der Prozedur sowieso niemanden mehr interessiert.

Beachten Sie bitte eine Besonderheit, die den Zugriff auf dynamische Arrays betrifft. Angenommen, Sie übergeben einer Assembler-Routine ein Element eines dynamischen Integerarrays mit Call Demo(A%(3)). Theoretisch sollte es der Routine möglich sein, auch auf das Element <A%(4)> zuzugreifen,

indem einfach der Wert 2 (Integervariablen werden als 2 Byte gespeichert) zum übergebenen Offset addiert wird.

Es ist nicht möglich! Denn wie Sie wissen, befindet sich das dynamische Array außerhalb von DS. Wir erhalten nur den Offset der Kopie von <A%(3)> in einer temporären Adresse in DS. Das heißt, auch lesend können wir nur auf dieses eine Element zugreifen, genauer: auf die Kopie dieses Elementes.

Auf komplette dynamische Arrays lesend oder schreibend zuzugreifen, erfordert ein etwas umständliches Vorgehen. Wir benötigen dazu die vollständige Arrayadresse, Segment und Offset. Wie wir diese absolute Adresse erhalten, zeige ich Ihnen bei der Erläuterung der Routine WINDOW, die genau dieses Problem zu bewältigen hat (WINDOWS greift auf das dynamische Integerarray <Scr%(...)> zu).

Fassen wir zusammen: QuickBASIC übergibt immer Offsets auf das Datensegment. Bei der Übergabe nach Referenz den Offset der übergebenen Variablen. Bei Übergabe nach Wert oder der Übergabe von Ausdrücken den Offset einer temporären Adresse, die eine Kopie der Variablen oder das Ergebnis des Ausdrucks enthält. Und den Zugriff auf Arrays besprechen wir noch.

Die Offsets bringt QuickBASIC als erstes auf den Stack, also noch vor der Rücksprungadresse. Ganz oben befindet sich der Offset des ersten Arguments, dann folgt der Offset des zweiten Arguments und so weiter.

	Argument Ø	Von QuickBASIC übergebene Argumente (Offsets zum Datensegment)
	Argument 1	
	.....	
[BP+12]	.....	
[BP+1Ø]	Argument N	
[BP+8]	Rücksprungadresse: Segment	Rücksprungadresse
[BP+6]	Rücksprungadresse: Offset	
[BP+4]	Gepushter Inhalt von BP	Diese Register müssen nach der Rückkher zu QuickBASIC unverändert sein !!!
[BP+2]	Gepushter Inhalt von SS	
BP =>	Gepushter Inhalt von DS	
[BP-2]	Temporäre Variable Ø	Eigene in der Assembler-Routine verwendete Variablen (je 2 Byte = 1 Wort)
[BP-4]	Temporäre Variable 1	
	.....	
	.....	
SP =>	Temporäre Variable 1Ø	

Relativ zum Basepointer adressieren Sie das letzte übergebene Argument mit [BP+10], das vorletzte mit [BP-12] und so weiter.

In Kürze geht es los. Wir besprechen PRINTF und WINDOWS. Aber zuvor noch eine kleine Übung. Wir schreiben eine Assembler-Routine, der ein String übergeben wird. Und diesen String soll die Routine ausgeben.

Zuerst muß ich Sie ein wenig »aufklären«. Bei der Speicherung von Strings legt QuickBASIC nicht einfach irgendwo die einzelnen Zeichen ab. Zusätzlich werden in einer Tabelle im Datensegment Informationen über jeden String gespeichert, die »Stringdescriptoren« (laut Handbuch »Zeichenfolgebeschreiber«; entsetzlicher Ausdruck, finden Sie nicht?).

Stringdescriptoren bestehen immer aus vier Byte. Die beiden ersten Bytes enthalten die Stringlänge im Format Low-Byte/High-Byte und die beiden letzten Bytes die Stringadresse, also den Offset zum Datensegment.

```
Stringdescriptoren:      Byte Ø  Byte 1  Byte 3  Byte 4
                         └──────┘        └──────┘
                         Stringlänge     Stringadresse
```

Beim Aufruf mit einem Stringargument übergibt QuickBASIC einen Zeiger (Offset auf das Datensegment) auf die Stringdescriptoren und nicht auf den String selbst. Da die Descriptoren die Stringadresse enthalten, können wir anschließend auf den String selbst zugreifen.

übergebenes Argument => Stringdescriptoren => String

Dieses Spielchen der »doppelten Verpointerung« kennt wohl jeder Assembler-Programmierer zur Genüge. Ein Pointer weist auf einen weiteren Pointer, der auf die eigentlich interessierende Adresse zeigt.

Angenommen, beim Aufruf wird der Routine nur ein Argument übergeben, eben der String. Dann finden wir dieses Argument relativ zu BP bei [BP+10], unmittelbar vor der Rücksprungadresse (siehe Schema).

Wir bringen den Inhalt von [BP+10] nach BX. BX enthält nun die Adresse (den Offset zum Datensegment) der Stringdescriptoren. Die Adresse BP enthält die Stringlänge und BP+2 die Adresse, an der sich der String befindet. Wir können ohne größere Probleme zum Beispiel das erste Zeichen des Strings nach AL bringen.

```
Mov  BX,[BP+1Ø]  ;Adresse der Descriptoren nach BX
Mov  CX,[BX]     ;Stringlänge nach CX
Mov  SI,[BX]+2   ;Stringadresse nach SI
Mov      AL,[SI]     ;Erstes Zeichen nach AL
```

Das »Zugriffschema« kennen Sie. Das Programm zur Ausgabe eines Strings geht nach diesem Schema vor. Allerdings wird nicht nur ein Zeichen nach AL kopiert, sondern der gesamte String wird mit der DOS-Funktion »Zeichenausgabe« (Int21H, Funktion 6) ausgegeben.

Zuvor zeige ich Ihnen noch, wie Sie auf Integervariablen zugreifen. Bei einem Aufruf wie *Call Demo(A%)* übergibt QuickBASIC auf dem Stack die Adresse der Integervariablen (Offset zum Datensegment). Und zwar nicht auf irgendeinen Descriptor, sondern direkt auf den Wert. Der Zugriff auf Integervariablen ist also sehr einfach. Sie greifen relativ zu BP auf das übergebene Argument zu, zum Beispiel auf [BP+10], wenn es sich um das letzte übergebene Argument handelt. Nun kennen Sie den Offset und können direkt auf den Integerwert zugreifen.

```
Mov  BX,[BX+1Ø]  ;Offset der Integervariablen holen
Mov  BX,[BX]     ;Integerwert holen
```

Und nun kümmern wir uns um STROUT.

```
;STROUT      Aufruf: Call Strout(<Stringausdruck>)
;------
WorkSpace    equ 2Ø              ;Platz für lokale Variablen
Argument     equ [BP+1Ø]
```

```
Data        Segment Word Public 'Data'
Data        Ends
DGroup      Group Data

Code        Segment Byte Public 'Code'
            Assume CS:Code, DS:DGroup

            Public Strout
Strout      Proc Far

; ----- Vorbereitungen -----
            Push BP             ;Register retten
            Push SS
            Push DS
            Mov  BP,SP          ;BP = SP = 1.Parameter – 12
            Sub  SP, WorkSpace  ;Platz für lokale Variablen

; ----- Stringlaenge und Stringadresse holen -----
            Mov  BX,Argument    ;Offset nach BX
            Mov  CX,[BX]        ;CX = Länge   (Inhalt von BX)
            Mov  SI,[BX]+2      ;SI = Adresse (Inhalt von BX + 2)
            Mov  AH,6           ;Funktion 6 von Interrupt 21H

; ----- String ausgeben -----
Ausgabe:    Mov  DL,[SI]        ;Char an Adresse DS:SI holen
            Int  21H            ;und ausgeben
            Inc  SI             ;inkrem. => zeigt auf next Char
            Loop Ausgabe        ;CX dekrementieren, nicht null =>

; ----- Stack korrigieren/Register wiederherstellen -----
EndLoop:    Mov  SP, BP         ;Reservierung aufheben
            Pop  DS
            Pop  SS
            Pop  BP
            Ret  2              ;Stack korrigieren

Strout      Endp
Code        Ends
            End
```

Nach dem Aufruf – zum Beispiel mit Call Strout("Test") – befindet sich das übergebene Argument (die Adresse der Descriptoren) relativ zu BP bei [BP+10] (Argument equ [BP+10]). Wie erläutert, wird die Stringlänge nach CX und die Adresse nach SI gebracht. SI zeigt nun auf die Startadresse des Strings.

In der Schleife wird jeweils das Zeichen, auf das SI zeigt, nach DL gebracht und ausgegeben. SI wird inkrementiert und weist nun auf das nächste Zeichen. Loop dekrementiert den Zähler CX (Stringlänge) und verzweigt zum Schleifenanfang, wenn noch nicht alle Zeichen ausgegeben sind.

Auf den ersten Blick fällt Ihnen der Unterschied zu den vorigen Programmen vielleicht nicht auf: Statt mit Ret wird das Programm mit Ret 2 beendet.

RET holt die Rückkehradresse vom Stapel und lädt den Befehlszeiger IP mit diesem Wert. Das heißt, das Programm wird an der von QuickBASIC auf den Stack gebrachten Rückkehradresse fortgesetzt.

Aber außer der Rückkehradresse brachte QuickBASIC ein Argument auf den Stack, also einen 2-Byte-Offset! Dieser Offset muß ebenfalls wieder vom Stack »heruntergezogen« werden. Genau das ist die Aufgabe von RET N. RET N korrigiert den Stackpointer um N Bytes. N ist gleich der doppelten Anzahl an übergebenen Argumenten, da für jedes Argument ein 2-Byte-Offset auf den Stack gebracht wird.

In unserem Fall wird ein Argument übergeben, ein 2-Byte-Offset. Daher verwendet das Programm Ret 2 – um den Stack um diese 2 Byte zu korrigieren!

11.5 Schnelle Bildschirmausgabe: PRINTF

Beim Aufruf von PRINTF übergeben wir eine ganze Menge Parameter.

```
Call PrintF(<Spalte>, <Zeile>, <Stringausdruck>, <Attribut>)
```

Beachten Sie bitte die Reihenfolge. <Spalte> ist das erste Argument und liegt ganz oben auf dem Stack. Als »unterstes« Argument übergibt QuickBASIC <Attribut>.

[BP+16]	Spalte
[BP+14]	Zeile
[BP+12]	Stringausdruck
[BP+10]	Attribut
	.....
	.....

Entsprechend werden am Anfang von PRINTF entsprechende Label deklariert.

```
;PRINTF
;------
;Funktion: Schnelle PRINT-Routine
;Aufruf: Call PrintF(Col%, Row%, String$, Color%)
;        Col%/Row% : Ausgabestart
;        String$   : Ausgabestring
;        Color%    : Ausgabefarbe
```

```
; ----- interne Label -----
WorkSpace     equ 2Ø              ;Platz für lokale Variablen
MonoCard      equ ØBØØØH          ;Monochrom-Karte (Hercules)
ColorCard     equ ØB8ØØH          ;Farb-Karte (CGA/EGA)
Card          equ [BP-2]          ;Flag für Bildschirmadapter

; ----- übergebene Parameter -----
Col           equ [BP+16]
Row           equ [BP+14]
String        equ [BP+12]
Color         equ [BP+1Ø]

VideoStat     equ Ø3DAH

Data          Segment Word Public 'Data'
Data          Ends
DGroup        Group Data

Code          Segment Byte Public 'Code'
              Assume CS:Code, DS:DGroup

              Public PrintF
PrintF        Proc Far
```

<MonoCard> und <ColorCard> sind die Startadressen eines monochromen (Monochromadapter/Herculeskarte) beziehungsweise eines Farbadapters (CGA/EGA). Bedenken Sie: PRINTF schreibt direkt in den Bildschirmspeicher! In <Flag> werden wir Informationen über den verwendeten Bildschirmadapter speichern.

<VideoStat> ist ein Register des Videoadapters. Wir werden im folgenden zwischen der Ausgabe auf einen monochromen und einen Farbadapter unterscheiden. Wenn Sie mit der CGA-Karte arbeiten, kennen Sie sicher den berüchtigten »Schnee-Effekt«.

Dieser Effekt kommt zustande, wenn »unkontrolliert« auf den Bildschirmspeicher der CGA-Karte zugegriffen wird. Ohne Störungen sind Zugriffe nur möglich, wenn eine Art »Freigabesignal« kommt. Und dieses Freigabesignal wird über das Register 03DAH übermittelt.

PRINTF enthält zwei verschiedene Ausgaberoutinen. Eine für die Ausgabe auf einen monochromen Adapter, die einfach Zeichen für Zeichen so schnell wie möglich in den Bildschirmspeicher schreibt.

Die Routine zur Ausgabe auf einen Farbadapter ist leider erheblich langsamer. Wenn das Freigabesignal kommt, wird genau ein Zeichen (plus Attribut) in den Bildschirmspeicher übertragen. Vor der Ausgabe des nächsten Zeichens wird erneut auf die Freigabe gewartet.

Dieses Warten auf das Freigabesignal nimmt weitaus mehr Zeit in Anspruch als die eigentliche Zeichenausgabe. Daher ist die Farbausgabe unglaublich

langsam im Vergleich zur Ausgabe auf einen monochromen Adapter. Aber immer noch sehr viel schneller als unter Verwendung der PRINT-Anweisung, da die extrem langsamen BIOS- und DOS-Funktionen umgangen werden.

Den nächsten Abschnitt kennen Sie. Die Registerinhalte werden gerettet und Platz für unsere Variablen reserviert.

```
; ----- Vorbereitungen -----
            Push BP              ;Register retten
            Push SS
            Push DS
            Mov  BP,SP           ;BP = SP = 1.Parameter – 12
            Sub  SP, WorkSpace   ;Platz für lokale Variablen
```

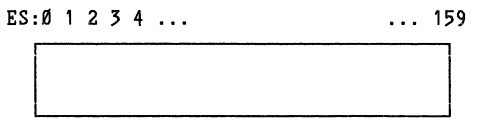
Und nun wird's interessant. Das Extrasegment-Register ES soll auf die Startadresse des Bildschirm-Adapters zeigen. Die hängt jedoch vom Adaptertyp ab. Über den Interrupt 11H wird die BIOS-Funktion »Equipment« aufgerufen, die unter anderem über den angeschlossenen Adapter Auskunft gibt.

```
; ----- Test auf Monochrom-/Farbadapter -----
            Int  11H
            And  AL,ØØ11ØØØØB  ;nur Bit 4 und 5 interessieren
            Mov  Card,AX       ;Card: Flag für Adaptertyp
            Cmp  AL,ØØ11ØØØØB  ;Bit 4 und 5 gesetzt?
            Jne  Farbe         ;Nein => Farbadapter
            Mov  AX,MonoCard   ;Ja   => Monochromadapter
            Jmp  GetPtr
Farbe:      Mov  AX,ColorCard
```

Nach Aufruf des Interrupts enthält AX den »Ausrüstungsstatus«. Wenn Bit 4 und 5 gesetzt sind, ist ein monochromer Adapter angeschlossen.

Alle anderen Bits werden ausmaskiert, das Resultat in <Card> gespeichert und geprüft, ob Bit 4 und 5 gesetzt sind. Wenn ja, wird die Startadresse des Bildschirmspeichers der Monochromadapter nach AX übertragen, sonst die entsprechende Adresse der Farbadapter. Im nächsten Abschnitt wird der Inhalt von AX nach ES übertragen – das Extrasegment ist initialisiert.

Die Startadresse des Bildschirmspeichers entspricht der linken oberen Bildschirmecke.



ES:0 enthält das erste Zeichenbyte, ES:1 das zugehörige Attributbyte, ES:2 das zweite Zeichenbyte, ES:3 das dritte Zeichenbyte und so weiter. Zeichen-

und Attributbyte befinden sich also unmittelbar nebeneinander, bilden ein »Wort«.

Die Initialisierung von ES genügt noch nicht. Wo die Ausgabe stattfindet, hängt von den Parametern <Col> und <Row> ab. Wir benötigen also noch einen Offset zum Extrasegment. Und dieser Offset wird nun in Abhängigkeit von <Col> und <Row> berechnet.

```
; ----- Zeiger auf Startposition nach ES:SI -----
GetPtr:     Mov  ES,AX          ;ES = Screenanfang
            Mov  BX,Row
            Mov  BX,[BX]
            Dec  BX
            Mov  AX,160
            Mul  BX
            Mov  BX,Col
            Mov  BX,[BX]
            Dec  BX
            Add  AX,BX
            Add  AX,BX
            Mov  DI,AX
```

Die Funktionsweise dieses etwas umfangreicheren Programmabschnitts:

1. Mov ES,AX initialisiert das Extrasegment mit der ermittelten Startadresse.
2. Mit Mov BX,Row und Mov BX,[BX] wird die übergebene Ausgabezeile nach BX gebracht. Anschließend wird die Zeile dekrementiert (Zeile ab 0 zählen) und multipliziert. Mit 160, da jede Zeile 160 Byte entspricht (80 Zeichenbyte plus 80 Attributbyte).
3. Mov BX,Col und Mov BX,[BX] holen die Startspalte nach BX. BX wird ebenfalls dekrementiert (Spalte ab 0 zählen) und anschließend zum bisherigen Offset der doppelte Wert von BX addiert (doppelt, um die Attributbytes zu berücksichtigen).
4. Mov DI,AX überträgt das Ergebnis, den Offset zum Extragsegment, in den »Destination-Index« DI. ES:SI zeigt auf die Ausgabeposition des ersten Zeichens.

Der nächste Abschnitt greift auf die Stringlänge und die Stringadresse zu.

```
; ----- Stringlaenge und Stringadresse holen -----
            Mov  BX,String
            Mov  CX,[BX]        ;CX = Laenge
            Mov  SI,[BX]+2      ;DX = Offset Adresse
            Mov  BX,Color
            Mov  BH,[BX]        ;BH = Attribut
```

Den ersten Teil kennen Sie inzwischen gut genug. Die Stringlänge wird nach CX und die Adresse nach SI (»Source-Index«) kopiert.

Danach wird der letzte Parameter geholt, <Attribut>. Dieser Integerwert bestimmt den Ausgabemodus, das Attributbyte. Dieses Attributbyte wird sich während der gesamten Ausgabeschleife in BH befinden.

```
            Cli                     ;Interrupts sperren

            Mov  DX,VideoStat
            Mov  AX,Card
            Cmp  AL,ØØ11ØØØØB
            Jne  RetLoop
```

Nachdem aus Geschwindigkeitsgründen Interrupts gesperrt werden, kommt die Adresse des Ports <VideoStat> nach DX und <Card>, das Flag für den Adaptertyp, wird erneut getestet. Sind die Bits 4 und 5 nicht gesetzt, verzweigt das Programm zu RETLOOP, der Ausgabe auf einen Farbadapter. Sonst wird in der nun folgenden Schleife der String auf einem Monochromadapter ausgegeben.

```
; ----- String ohne Warten ausgeben -----
PrintLoop:  Mov  BL,[SI]
            Mov  ES:[DI],BX
            Inc  SI
            Add  DI,2
            Loop PrintLoop
            Jmp  EndLoop
```

BH enthält das Attributbyte. Das auszugebende Zeichen DS:[SI] wird nach BL übertragen. Die niederwertige Hälfte von BX enthält nun das Zeichen, die höherwertige das Attribut. BX wird in den Bildschirmspeicher übertragen, an die aktuelle Adresse ES:[DI].

Nun wird SI, der Zeiger auf den String, inkrementiert, und DI, der Zeiger auf den Bildschirm, um zwei erhöht. Die Ausgabe des nächsten Zeichens ist vorbereitet.

Zuvor wird jedoch geprüft, ob bereits alle Zeichen ausgegeben sind. *Loop PrintLoop* dekrementiert den Zähler CX (enthält die Stringlänge) und verzweigt zum Schleifenanfang, wenn das Ergebnis ungleich 0 ist.

Sonst sind wir fertig und mit Jmp EndLoop wird zum Programmende gesprungen, um die folgende »Farbschleife« zu überspringen.

```
; ----- String mit Warten auf Retrace ausgeben -----
Wait1:      In   AL,DX
            Test AL,1
            Jnz  Wait1
```

```
RetLoop:    Mov  BL,[SI]
Wait2:      In   AL,DX
            Test AL,1
            Jz   Wait2
            Mov  ES:[DI],BX
            Inc  SI
            Add  DI,2
            Loop RetLoop
```

Die Ausgabe auf einen Farbadapter unterscheidet sich nur in einem Punkt vom vorigen Ablauf: Bevor das Zeichen in den Bildschirmspeicher geschrieben wird, wartet die Routine auf das Freigabesignal. Die beiden Testschleifen erläutere ich nicht näher. Ich muß zugeben: Ich habe sie abgeschrieben (aus dem BIOS-ROM). Also finden wir uns einfach mit den Befehlen ab und freuen uns, daß es keinen »Schneefall« auf dem Bildschirm gibt. Aber denken Sie daran: Die Warteschleifen setzen voraus, daß DX die Adresse <VideoStat> enthält!

Und nun wird es noch einmal interessant. Wir sind am Programmende, beim Wiederherstellen des Ausgangszustands des Stacks und der Register.

```
; ----- Stack korrigieren/Register wiederherstellen -----
EndLoop:    Sti                 ;Interrupts zulassen
            Mov  SP, BP         ;Reservierung für lokale Variablen aufhe-
ben
            Pop  DS
            Pop  SS
            Pop  BP
            Ret  8              ;Stack gemäß Parameteranzahl korrigieren

PrintF      Endp
Code        Ends
            End
```

Achten Sie bitte auf Ret 8. Vier Argumente wurden übergeben. Viermal brachte QuickBASIC je einen 2-Byte-Offset auf den Stack. Nach der Rückkehr ist der Stackpointer um 2 * 4 = 8 Byte zu korrigieren!

11.6 Listing von PRINTF.ASM

```
;PRINTF
;------
;Funktion: Schnelle PRINT-Routine
;Aufruf: Call PrintF(Col%, Row%, String$, Color%)
;        Col%/Row% : Ausgabestart
;        String$   : Ausgabestring
;        Color%    : Ausgabefarbe

; ----- interne Label -----
WorkSpace    equ 20              ;Platz für lokale Variablen
MonoCard     equ 0B000H          ;Monochrom-Karte (Hercules)
ColorCard    equ 0B800H          ;Farb-Karte (CGA/EGA)
Card         equ [BP-2]          ;Flag für Bildschirmadapter

; ----- übergebene Parameter -----
Col          equ [BP+16]
Row          equ [BP+14]
String       equ [BP+12]
Color        equ [BP+10]

VideoStat    equ 03DAH

Data         Segment Word Public 'Data'
Data         Ends
DGroup       Group Data

Code         Segment Byte Public 'Code'
             Assume CS:Code, DS:DGroup

             Public PrintF
PrintF       Proc Far

; ----- Vorbereitungen -----
             Push BP             ;Register retten
             Push SS
             Push DS
             Mov  BP,SP          ;BP = SP = 1.Parameter - 12
             Sub  SP, WorkSpace  ;Platz für lokale Variablen

; ----- Test auf Monochrom-/Farbadapter -----
             Int  11H
             And  AL,00110000B  ;nur Bit 4 und 5 interessieren
             Mov  Card,AX       ;Card: Flag für Adaptertyp
             Cmp  AL,00110000B  ;Bit 4 und 5 gesetzt?
             Jne  Farbe         ;Nein => Farbadapter
             Mov  AX,MonoCard   ;Ja   => Monochromadapter
             Jmp  GetPtr
Farbe:       Mov  AX,ColorCard

; ----- Zeiger auf Startposition nach ES:SI -----
GetPtr:      Mov  ES,AX          ;ES = Screenanfang
             Mov  BX,Row
             Mov  BX,[BX]
             Dec  BX
             Mov  AX,160
```

```
            Mul  BX
            Mov  BX,Col
            Mov  BX,[BX]
            Dec  BX
            Add  AX,BX
            Add  AX,BX
            Mov  DI,AX

; ----- Stringlaenge und Stringadresse holen -----
            Mov  BX,String
            Mov  CX,[BX]        ;CX = Laenge
            Mov  SI,[BX]+2      ;DX = Offset Adresse
            Mov  BX,Color
            Mov  BH,[BX]        ;BH = Color

            Cli                 ;Interrupts sperren

            Mov  DX,VideoStat
            Mov  AX,Card
            Cmp  AL,00110000B
            Jne  Wait1

; ----- String ohne Warten ausgeben -----
PrintLoop:  Mov  BL,[SI]
            Mov  ES:[DI],BX
            Inc  SI
            Add  DI,2
            Loop PrintLoop
            Jmp  EndLoop

; ----- String mit Warten auf Retrace ausgeben -----
Wait1:      In   AL,DX
            Test AL,1
            Jnz  Wait1

RetLoop:    Mov  BL,[SI]
Wait2:      In   AL,DX
            Test AL,1
            Jz   Wait2
            Mov  ES:[DI],BX
            Inc  SI
            Add  DI,2
            Loop RetLoop

; ----- Stack korrigieren/Register wiederherstellen -----
EndLoop:    Sti                 ;Interrupts zulassen
            Mov  SP, BP         ;Reservierung aufheben
            Pop  DS
            Pop  SS
            Pop  BP
            Ret  8              ;Stack korrigieren

PrintF      Endp
Code        Ends
            End
```

11.7 Windowing: WINDOWS

Der Aufbau von WINDOWS ist PRINTF sehr ähnlich. Kein Wunder, denn WINDOWS entstand durch Erweiterung und Umbau von PRINTF. Diesmal gibt jedoch vier Schleifen, von denen genau eine durchlaufen wird.

Zwei Schleifen behandeln Monochromkarten. Eine Schleife rettet einen Bildschirmausschnitt, eine zweite kopiert einen geretteten Ausschnitt auf den Bildschirm zurück.

Zwei weitere Schleifen machen genau das Gleiche, behandeln jedoch das Retten/Holen mit einer Farbkarte. Das heißt, Sie warten vor jedem Zugriff auf den Bildschirmspeicher auf das Freigabesignal.

Nun fragen Sie sich bestimmt, warum dieser Aufwand nötig ist. Theoretisch wäre mit einer eigenen Prozedur »Warten auf Freigabesignal« viel Schreibarbeit einzusparen. Dann sollten eigentlich zwei Schleifen ausreichen, eine zum Retten, eine zum Holen eines Ausschnitts.

In beiden Schleifen wird vor jedem Zugriff auf den Bildschirmspeicher der Adaptertyp getestet. Ist es ein Farbadapter, rufen wir einfach die Prozedur »Warten auf Freigabesignal« auf.

Genau das waren auch meine Vorstellungen. Leider scheiterten Sie an der Geschwindigkeit des Elektronenstrahls, der über den Bildschirm flitzt. Das Freigabesignal wird gegeben, wenn sich dieser Strahl an einer bestimmten Position befindet. Dann ist für einen winzigen Moment der Zugriff auf den Bildschirmspeicher ohne Bildstörungen möglich.

Bei meinen Versuchen stellte ich fest, daß dieser Moment so extrem kurz ist, daß der Zugriff sofort nach der Freigabe erfolgen muß. »Sofort« heißt, daß selbst die Ausführung eines RET-Befehl zur Rückkehr aus der Prozedur zuviel Zeit benötigt. Auf die Freigabe muß der MOV-Befehl kommen, ohne dazwischenliegende Verzögerungen.

Daher enthält das Programm keine Prozedur »Warten« und ist so umständlich in zwei nahezu identische Abschnitte aufgeteilt.

```
;WINDOWS
;-------
;Funktion: rettet/holt Bildschirmausschnitt in/aus Integerarray
;Aufruf: Call Windows(Col%,Row%,Breite%,Laenge%,VarSeg%,VarOff%,Flag%)
;        Col%/Row%          : Linke obere Ecke
;        Breite%            : Breite in Spalten
;        Laenge%            : Laenge in Zeilen
;        VarSeg%/VarOff%    : Adresse eines Arrayelements (VARPTR+PTR86)
;        Flag%              : Ø = retten / 1 = holen
```

```
; ----- interne Label -----
WorkSpace     equ 20          ;Platz für lokale Variablen auf Stack
MonoCard      equ 0B000H      ;Adresse Monochrom-Karte (Hercules)
ColorCard     equ 0B800H      ;Adresse Farb-Karte (CGA/EGA)
Spalten       equ [BP-2]      ;Word Ptr: Zähler Innenschleife (Breite)
Zeilen        equ [BP-4]      ;Word Ptr: Zähler Außenschleife (Länge)

; ----- übergebene Parameter -----
Col           equ [BP+22]     ;Spalte linke obere Ecke
Row           equ [BP+20]     ;Zeile linke obere Ecke
Breite        equ [BP+18]     ;Breite in Spalten
Laenge        equ [BP+16]     ;Länge in Zeilen
VarSeg        equ [BP+14]     ;Arrayadresse: Segment
VarOff        equ [BP+12]     ;Arrayadresse: Offset
Flag          equ [BP+10]     ;0 = retten / 1 = holen

VideoStat     equ 03DAH       ;Videocontroller: Retrace-Signal-Status


Data          Segment Word Public 'Data'
Data          Ends
DGroup        Group Data

Code          Segment Byte Public 'Code'
              Assume CS:Code, DS:DGroup

              Public Windows
Windows       Proc Far


; ------------------------- INITIALISIERUNG -----------------------
              Push BP             ;Register retten
              Push SS
              Push DS
              Mov  BP,SP
              Sub  SP,WorkSpace   ;Platz für lokale Variablen
```

Kommt Ihnen dieser »Vorspann« bekannt vor? Auch hier treffen Sie übliche Programmabschnitte an, die Deklaration der Label, der Segmente, das Retten der Register und die Reservierung eines geschützten Stackbereichs für unsere Variablen. Also kümmern wir uns nicht weiter um diesen Teil.

Damit der nächste Abschnitt (Adaptertest) nicht ebenso langweilig wird, wird das Adapterflag im Gegensatz zu PRINTF nicht auf dem Stack gespeichert (in <Card>), sondern zur Abwechslung im Register CL.

```
; ----- Test auf Monochrom-/Farbadapter -----
              Int  11H            ;testet Equipment
              And  AL,48          ;nur Bit 4 und 5 interessieren
              Mov  CL,AL          ;Ausmaskiertes Flag in CL retten
              Cmp  CL,48          ;Bit 4 und 5 gesetzt?
              Jne  ColorAdress    ;Nein : Farbadapter, Sprung =>
              Mov  AX,MonoCard    ;Ja   : Monochromadapter
              Jmp  GetPtr
ColorAdress:  Mov  AX,ColorCard
```

Davon abgesehen, wurde dieser Teil unverändert aus PRINTF übernommen. Ebenso wie der nächste, der in ES:SI einen Zeiger auf die linke obere Ecke des Ausschnitts erzeugt.

```
; ----- Zeiger auf Startposition nach ES:SI -----
GetPtr:     Mov  ES,AX          ;ES = Adresse der Bildschirm-Karte
            Mov  BX,Row         ;SI ergibt sich aus:
            Mov  BX,[BX]        ;(Zeilenanzahl - 1) * 160
            Dec  BX             ;(160 Byte pro Zeile)
            Mov  AX,160         ;plus
            Mul  BX             ;(Spaltenanzahl - 1) * 2
            Mov  BX,Col
            Mov  BX,[BX]
            Dec  BX
            Add  AX,BX
            Add  AX,BX
            Mov  SI,AX
```

Und nun kommt endlich mal ein neuer Teil. Die Parameter <Flag>, <Zeilen> und <Spalten> sind Integervariablen. Mit der erläuterten Methode (auf den Offset relativ zu BP zugreifen, dann den Inhalt der Adresse holen) wird nacheinander auf jede dieser Variablen zugegriffen.

<Flag> kommt nach CH (CL ist bereits mit dem Adaptertyp belegt). Die Ausschnittlänge und -breite wird in unserem geschützten Stackbereich gespeichert, in <Zeilen> und <Spalten>.

```
; ----- Richtungsflag holen / Schleifenzähler initialisieren -----
            Mov  BX,Flag
            Mov  CH,[BX]        ;CH = Richtungs (0=retten / 1=holen)

            Mov  BX,Laenge      ;Zeilen = Anzahl der zu
            Mov  BX,[BX]        ;behandelnden Zeilen
            Mov  Zeilen,BX

            Mov  BX,Breite      ;Spalten = Anzahl der zu behandelnden
            Mov  BX,[BX]        ;Bytes pro Zeile (= Breite * 2)
            Add  BX,BX
            Mov  Spalten,BX
```

Nun kommt der schwierigste Programmteil. DS:DI soll auf das Integerarray zeigen. Der Programmteil selbst ist harmlos, der »Hintergrund« jedoch etwas verwickelt.

Erinnern Sie sich: Zum Retten eines Ausschnitts verwenden wir das dynamische Array <Scr%(..)>. Dynamische Arrays befinden sich außerhalb von DS, also nicht im Datensegment von QuickBASIC. Wird beim Aufruf ein Element dieses Arrays angegeben, kopiert QuickBASIC dieses Element in eine temporäre Adresse in DS und übergibt uns den Offset dieser Adresse.

Leider können wir mit diesem einen Element recht wenig anfangen. »Retten eines Ausschnitts«, das heißt, wir benötigen eine ganze Menge an Array-

elementen, um den Ausschnitt darin unterzubringen. Also müssen wir auf das Array selbst zugreifen, nicht nur auf die Kopie eines Elementes.

Um diese Problematik kümmert sich zum Glück das aufrufende BASIC-Programm (PUSHSCREEN und PULLSCREEN). Es ermittelt mit VARPTR die Adresse des »Startelements«, ab dem der Ausschnitt gespeichert werden soll.

Anschließend ruft es PTR86 auf. PTR86 wandelt die von VARPTR übergebene Adresse in einen Segment-/Offsetwert. Und diese Segment-/Offsetadresse übergibt uns PULLSCREEN und PUSHSCREEN beim Aufruf.

Der folgende Programmteil hat dank dieser Vorbereitungen eine recht einfache Aufgabe. Er liest die Inhalte der übergebenen Variablen und <Offset> und kopiert sie in DS und DI. Zum Abschluß werden Interrupts gesperrt, um die Geschwindigkeit der folgenden Ausgabeschleife nicht unnötig zu bremsen.

```
; ----- Zeiger auf Integerarray nach DS:DI -----
            Mov  BX,VarOff
            Mov  DI,[BX]
            Mov  BX,VarSeg
            Mov  DS,[BX]

            Cli                 ;Interrupts sperren
```

Halten wir fest: ES:SI zeigt auf den Beginn des Bildschirmausschnitts, der gerettet/wiederhergestellt werden soll. DS:DI zeigt auf das Arrayelement, ab dem der Ausschnitt in das Array zu kopieren ist (beziehungsweise, ab dem der wiederherzustellende Ausschnitt gespeichert ist).

Nachdem alle Vorbereitungen getroffen sind, wird der Adaptertyp getestet.

```
; ----- Farbe oder Monochrom? -----
            Cmp  CL,48          ;Monochrom?
            Jne  ColLoop        ;Nein, dann Sprung =>
```

Nach dem Aufruf von INT21H wurde der Ausrüstungsstatus nach CL übertragen (zuvor wurden alle Bits außer 4 und 5 ausmaskiert). Wenn in CL zumindest eines der beiden Bits 4 und 5 nicht gesetzt ist, ist kein Monochromadapter angeschlossen. Das Programm verzweigt zu COLLOOP.

```
; ----- Untergrund retten oder holen? -----
            Cmp  CH,Ø           ;Ausschnitt retten?
            Jne  MonoGet        ;Nein, dann Sprung =>
```

CH enthält das »Richtungsflag«. 0 bedeutet, daß ein Ausschnitt gerettet werden soll. In diesem Fall wird die folgende Schleife abgearbeitet. Sonst verzweigt das Programm zu MONOGET, dem Zurückkopieren eines bereits geretteten Ausschnitts.

```
; ----- Bildschirm => Integerarray -----
MonoPut:     Mov  BX,Ø
MonoPut1:    Mov  CX,ES:[SI][BX]
             Mov  DS:[DI],CX
             Add  DI,2
             Add  BX,2
             Cmp  BX,Spalten
             Jne  MonoPut1
             Add  SI,16Ø
             Dec  Word Ptr Zeilen
             Jnz  MonoPut
             Jmp  Ende
```

Das Kopieren verläuft ähnlich wie die Stringausgabe in PRINTF. ES:SI zeigt auf die Startadresse des Ausschnitts. BX wird mit 0 initialisiert und nach jedem Durchgang um 2 erhöht. Das heißt, nach jedem Durchgang wird auf das nächste Zeichen plus Attribut zugegriffen.

Wo die Zeichen gespeichert werden, hängt vom Ausdruck DS:[DI] ab. DS:DI zeigt zu Beginn auf die übergebene Adresse innerhalb des verwendeten Integerarrays. Nach jedem Schleifendurchgang wird DI um 2 erhöht und zeigt damit auf das nächste Arrayelement.

Die innere Schleife ist beendet, wenn BX der Anzahl der zu kopierenden Spalten <Spalten> entspricht. Eine Windowzeile wurde gerettet. SI wird um 160 erhöht und zeigt nun auf das erste Zeichen der darunterliegenden Windowzeile.

<Zeilen> wird vermindert und geprüft, ob bereits alle Zeilen des Windows behandelt wurden (wenn <Zeilen> den Wert 0 enthält). Wenn nein, wird zum Anfang der äußeren Schleife verzweigt – die nächste Zeile wird behandelt.

MONOGET behandelt die umgekehrte Richtung, das Zurückschreiben eines geretteten Ausschnitts. Der Ablauf bleibt gleich, diesmal wird jedoch gerade umgekehrt aus jener Adresse gelesen, auf die DS:[DI] zeigt, und zu jener Adresse kopiert, auf die ES:[SI][BX] weist.

```
; ----- Integerarray => Bildschirm -----
MonoGet:     Mov  BX,Ø
MonoGet1:    Mov  CX,DS:[DI]
             Mov  ES:[SI][BX],CX
             Add  DI,2
             Add  BX,2
             Cmp  BX,Spalten
             Jne  MonoGet1
```

```
Add  SI,160
Dec  Word Ptr Zeilen
Jnz  MonoGet
Jmp  Ende
```

Die entsprechenden Schleifen zur Behandlung eines Farbadapters erspare ich Ihnen. Sie entsprechen den beiden letzten Programmteilen bis ins Detail, mit dem einen Unterschied, daß vor jedem Zugriff auf den Bildschirmspeicher das Freigabesignal abgewartet wird.

So, nun sollten Sie in der Lage sein, eigene Assembler-Routinen zu schreiben, die mit QuickBASIC zusammenarbeiten. Aber um ehrlich zu sein: Ich sehe keine allzu große Notwendigkeit dafür.

PRINTF und WINDOWS sind nötig, weil die Bildschirmausgabe über DOS- oder BIOS-Routinen einen echten Engpaß in Programmen darstellt. Davon abgesehen, ist die Ausführungszeit von QuickBASIC-Programmen eigentlich für fast jede Anwendung ausreichend. Auf alle Fälle besitzen Sie nun das notwendige Handwerkszeug und ich kann Ihnen nur noch viel Spaß wünschen.

11.8 Listing von WINDOWS.ASM

```
;WINDOWS
;-------
;Funktion: rettet/holt Bildschirmausschnitt in/aus Integerarray
;Aufruf: Call Windows(Col%,Row%,Breite%,Laenge%,VarSeg%,VarOff%,Flag%)
;        Col%/Row%          : Linke obere Ecke
;        Breite%            : Breite in Spalten
;        Laenge%            : Laenge in Zeilen
;        VarSeg%/VarOff%    : Adresse eines Arrayelements (VARPTR+PTR86)
;        Flag%              : 0 = retten / 1 = holen

;----- interne Label -----
WorkSpace    equ 20         ;Platz für lokale Variablen auf Stack
MonoCard     equ 0B000H     ;Adresse Monochrom-Karte (Hercules)
ColorCard    equ 0B800H     ;Adresse Farb-Karte (CGA/EGA)
Spalten      equ [BP-2]     ;Word Ptr: Zähler Innenschleife (Breite)
Zeilen       equ [BP-4]     ;Word Ptr: Zähler Außenschleife (Länge)

;----- übergebene Parameter -----
Col          equ [BP+22]    ;Spalte linke obere Ecke
Row          equ [BP+20]    ;Zeile linke obere Ecke
Breite       equ [BP+18]    ;Breite in Spalten
Laenge       equ [BP+16]    ;Länge in Zeilen
VarSeg       equ [BP+14]    ;Arrayadresse: Segment
VarOff       equ [BP+12]    ;Arrayadresse: Offset
Flag         equ [BP+10]    ;0 = retten / 1 = holen
```

```
VideoStat     equ Ø3DAH      ;Videocontroller: Retrace-Signal-Status


Data          Segment Word Public 'Data'
Data          Ends
DGroup        Group Data

Code          Segment Byte Public 'Code'
              Assume CS:Code, DS:DGroup

              Public Windows
Windows       Proc Far


;------------------------- INITIALISIERUNG -----------------------
              Push BP            ;Register retten
              Push SS
              Push DS
              Mov  BP,SP
              Sub  SP,WorkSpace  ;Platz für lokale Variablen

;----- Test auf Monochrom-/Farbadapter -----
              Int  11H           ;testet Equipment
              And  AL,48         ;nur Bit 4 und 5 interessieren
              Mov  CL,AL         ;Ausmaskiertes Flag in CL retten
              Cmp  CL,48         ;Bit 4 und 5 gesetzt?
              Jne  ColorAdress   ;Nein : Farbadapter, Sprung =>
              Mov  AX,MonoCard   ;Ja   : Monochromadapter
              Jmp  GetPtr
ColorAdress:  Mov  AX,ColorCard

;----- Zeiger auf Startposition nach ES:SI -----
GetPtr:       Mov  ES,AX         ;ES = Adresse der Bildschirm-Karte
              Mov  BX,Row        ;SI ergibt sich aus:
              Mov  BX,[BX]       ;(Zeilenanzahl - 1) * 16Ø
              Dec  BX            ;(16Ø Byte pro Zeile)
              Mov  AX,16Ø        ;plus
              Mul  BX            ;(Spaltenanzahl - 1) * 2
              Mov  BX,Col
              Mov  BX,[BX]
              Dec  BX
              Add  AX,BX
              Add  AX,BX
              Mov  SI,AX

;----- Richtungsflag holen / Schleifenzähler initialisieren -----
              Mov  BX,Flag
              Mov  CH,[BX]       ;CH = Richtungs (Ø=retten / 1=holen)

              Mov  BX,Laenge     ;Zeilen = Anzahl der zu
              Mov  BX,[BX]       ;behandelnden Zeilen
              Mov  Zeilen,BX

              Mov  BX,Breite     ;Spalten = Anzahl der zu behandelnden
              Mov  BX,[BX]       ;Bytes pro Zeile (= Breite * 2)
              Add  BX,BX
              Mov  Spalten,BX
```

```
;----- Zeiger auf Integerarray nach DS:DI -----
            Mov  BX,VarOff
            Mov  DI,[BX]
            Mov  BX,VarSeg
            Mov  DS,[BX]

            Cli                 ;Interrupts sperren

;----- Farbe oder Monochrom? -----
            Cmp  CL,48          ;Monochrom?
            Jne  ColLoop        ;Nein, dann Sprung =>
;-----------------------------------------------------------------


;--------------------- ZUGRIFF AUF MONOCHROM-KARTE -----------------

;----- Untergrund retten oder holen? -----
            Cmp  CH,Ø           ;Ausschnitt retten?
            Jne  MonoGet        ;Nein, dann Sprung =>

;----- Bildschirm => Integerarray -----
MonoPut:    Mov  BX,Ø
MonoPut1:   Mov  CX,ES:[SI][BX]
            Mov  DS:[DI],CX
            Add  DI,2
            Add  BX,2
            Cmp  BX,Spalten
            Jne  MonoPut1
            Add  SI,16Ø
            Dec  Word Ptr Zeilen
            Jnz  MonoPut
            Jmp  Ende

;----- Integerarray => Bildschirm -----
MonoGet:    Mov  BX,Ø
MonoGet1:   Mov  CX,DS:[DI]
            Mov  ES:[SI][BX],CX
            Add  DI,2
            Add  BX,2
            Cmp  BX,Spalten
            Jne  MonoGet1
            Add  SI,16Ø
            Dec  Word Ptr Zeilen
            Jnz  MonoGet
            Jmp  Ende
;-----------------------------------------------------------------


;------------------------ ZUGRIFF AUF FARB-KARTE -----------------

;----- Untergrund retten oder holen? -----
ColLoop:    Mov  DX,VideoStat
            Cmp  CH,Ø           ;Ausschnitt retten?
            Jne  ColGet         ;Nein, dann Sprung =>
```

```
;----- Bildschirm => Integerarray -----
;- Zugriffsverbot abwarten -
ColPut:       In   AL,DX          ;Auf Retrace-Signal warten,
              Test AL,1           ;dann erst Zugriff auf Video-RAM
              Jnz  ColPut

;- Zähler innere Schleife init.-
ColPut1:      Mov  BX,Ø

;- Warten, bis Retrace stattfindet -
ColPut2:      In   AL,DX
              Test AL,1
              Jz   ColPut2

; - Zeichen + Attribut lesen und schreiben -
              Mov  CX,ES:[SI][BX]
              Mov  DS:[DI],CX
              Add  DI,2
              Add  BX,2
              Cmp  BX,Spalten
              Jne  ColPut2
              Add  SI,16Ø
              Dec  Word Ptr Zeilen
              Jnz  ColPut1
              Jmp  Ende

;----- Integerarray => Bildschirm -----
;- Zugriffsverbot abwarten -
ColGet:       In   AL,DX          ;auf Retrace-Signal warten,
              Test AL,1           ;dann erst Zugriff auf Video-RAM
              Jnz  ColGet

;- Zähler innere Schleife init.-
ColGet1:      Mov  BX,Ø

; - Zeichen + Attribut lesen -
ColGet2:      Mov  CX,DS:[DI]

;- Warten, bis Retrace stattfindet -
ColGet3:      In   AL,DX
              Test AL,1
              Jz   ColGet3

;- Zeichen + Attribut schreiben -
              Mov  ES:[SI][BX],CX
              Add  DI,2
              Add  BX,2
              Cmp  BX,Spalten
              Jne  ColGet2
              Add  SI,16Ø
              Dec  Word Ptr Zeilen
              Jnz  ColGet1
;-----------------------------------------------------------------
```

```
;----- Stack korrigieren/Register wiederherstellen -----
Ende:       Sti                 ;Interrupts zulassen
            Mov  SP,BP
            Pop  DS
            Pop  SS
            Pop  BP
            Ret  14

Windows     Endp
Code        Ends
            End
```

Anhang

Programme auf der beiliegenden Diskette

Stammverzeichnis: Unterverzeichnisse und das ausführbare Programm CONVERT.EXE (GW-BASIC-Programme nach QuickBASIC konvertieren)

Verzeichnisse

- ASEMBLER : Assembler-Sources
- DEMOS : Kleine Demoprogramme
- BATCHES : Batch-Files zum Kompilieren und Linken
- MODULE : Module zur Einbindung in User-Librarys
- OBJFILES : Die fertig kompilierten Objektdateien

Dateien in ASEMBLER

- DEMO.ASM : 'A' ausgeben
- DEMO1.ASM : 'AB' ausgeben
- PRINTF.ASM : Schnelle PRINT-Routine
- STROUT.ASM : Zugriff auf Strings
- WINDOW.ASM : Windowing-Routine

Dateien in BATCHES

- DC.BAT : Kompilieren mit Diskettenlaufwerken
- DL.BAT : User-Library mit Diskettenlaufwerken erstellen
- PC.BAT : Kompilieren mit Festplatte
- PL.BAT : User-Library mit Festplatte erstellen

Dateien in DEMOS

- COMDEF.BAS : Deklaration modulübergreifender globaler Variablen
- COMMON.BAS : Deklarationen mit COMMON SHARED
- COMPRESS.BAS : Demo der COMPRESS-Routine
- FILL.BAS : Demo der FILL-Routine
- GDEMO.BAS : Korrekte Verwendung globaler Variablen in User-Libraries
- GLOBAL.BAS : Falsche Verwendung globaler Variablen in User-Libraries
- IF_BLOCK.BAS : Blockbildung mit IF..THEN
- INIT.BAS : Initialisierung der modulübergreifenden globalen Variablen
- INPUTDEM.BAS : Demo der LINPUT-Routine
- LABEL.BAS : Verwendung von Label statt Zeilennummern
- LIBRARY.BAS : Sammlung häufig benötigter Routinen
- LONG_IF.BAS : Mehrzeilige IF-Anweisungen
- LOWUP.BAS : Demo der beiden Prozeduren LOWUPCASE und UPLOWCASE
- MASKDEMO.BAS : Demo der Prozedur MASKE
- MAXWERT1.BAS : Größte Zahl in Array ermitteln, Version 1
- MAXWERT2.BAS : Größte Zahl in Array ermitteln, Version 2
- MAXWERT3.BAS : Größte Zahl in Array ermitteln, Version 3
- MWST.BAS : Mehrwertsteuer berechnen
- PRINTFDE.BAS : Demo der PRINTF-Routine
- PULLDEM.BAS : Demo der Routine PULLDOWN
- SOTHER.BAS : Demo von SEARCHOTHER
- STATIC.BAS : Anwendung statischer Variablen
- TASTE.BAS : Demo von TASTE
- USER1.BAS : Demo zum Aufbau von User-Libraries
- USER2.BAS : Demo zum Aufbau von User-Libraries
- USERDEF1.BAS : Demo zu benutzerdefinierten Funktionen
- USERDEF2.BAS : Demo zu benutzerdefinierten Funktionen
- WINDOWS.BAS : Demo zu den Windowing-Routinen

Dateien in MODULE

- COMDEF.BAS : Deklaration modulübergreifender globaler Variablen
- CONVERT.BAS : GW-BASIC-Programme nach QuickBASIC konvertieren
- INIT.BAS : Initialisierung der modulübergreifenden globalen Variablen
- INPUT.BAS : Eingaberoutine
- MASKE.BAS : Maskenverwaltung
- PULLDOWN.BAS : Verwaltung von Pull-down-Menüs
- ROUTINEN.BAS : Kleine Routinen-Sammlung; bis auf die zusätzlichen $INCLUDE-Anweisungen mit LIBRARY.BAS identisch

Dateien in OBJFILES

- CONVERT.OBJ : Kompiliertes Konvertierungsprogramm, zum Linken mit der Library BRUN20.EXE vorbereitet
- INPUT.OBJ : Kompilierte Eingaberoutine, vorbereitet zur Einbindung in eine User-Library
- INT86.OBJ : Assemblierte QuickBASIC-Routine; enthält die Prozedur PTR86, die Segment/Offset einer Variablen ermittelt
- MASKE.OBJ : Kompilierte Maskenverwaltung, vorbereitet zur Einbindung in eine User-Library
- PREFIX.OBJ : Assemblierte QuickBASIC-Routine; sollte beim Linken vor allen anderen Assembler-Programmen angegeben werden
- PRINTF.OBJ : Assemblierte schnelle Ausgaberoutine
- PULLDOWN.OBJ : Kompilierte Pull-down-Menü-Verwaltung, vorbereitet zur Einbindung in eine User-Library
- ROUTINEN.OBJ : Kompilierte Routinen-Sammlung, vorbereitet zur Einbindung in eine User-Library
- WINDOWS.OBJ : Assemblierte Windowing-Routine

Stichwortverzeichnis

$INCLUDE-Anweisung 70, 96 f., 135
$INCLUDE-Befehl 63 ff.
$INCLUDE-Files 83, 89

A
Ablaufgeschwindigkeit 19
Absolutbetrag 27
Adaptertyp 310, 312
Adresse 36, 55
Adreßübergabe 46
Adreßverwaltung 108, 142
Änderungs-Flag 129, 137
Anführungszeichen 18
Anweisungen 14, 18
Anweisungsblock 286
Arbeitsdiskette 85 ff., 93, 96, 101 f., 105 f., 128
Argumente 32, 145
Argumentliste 29, 30, 43, 47
Array 15, 46, 51, 181, 183, 206, 223, 229, 289, 297, 313
Array-Grenzfunktion 52
Arrayadresse 298
Arraydimension 51, 265
Arrayelement 219, 313
Arraygrenzen 20
Arraygröße 53, 186
Arrayname 53, 59 f.
Arrayvariable 14, 52
ASCII-Code 74, 295
ASCII-Datei 64
ASCII-Format 263
ASCII-Tabelle 72 f., 80
Assembler 289
Assembler-Module 276
Assembler-Programm 289
Assembler-Quelltext 290
Assembler-Routine 184, 195, 295
Attributbyte 305 f.
Ausdrücke 49, 297
Ausgabeoptionen 21, 100
Ausgabeschleife 136
Ausgangswerte 111
Austesten 20, 100, 104

B
Basepointer 299
BASIC-Quelltext 85 f.
Batch-Datei 101 ff.
Batch-Files 269
BCOM20.LIB 16, 273 ff.
Befehls-Stapel 103
Befehlsnummer ermitteln 181
Befehlswörter 251
Begriffsverwendung 14
Benutzer-Bibliotheken 15
Betriebssystem 104
Betriebssystem-Diskette 106
Betriebssystem-Ebene 87
Bibliothek 15, 32, 69
Bildschirm-Editor 107, 131, 133
Bildschirm-Koordinaten 107
Bildschirmadapter 289
Bildschirmausgabe 113, 144, 218, 289, 302, 315
Bildschirmausschnitt 173, 218, 310
Bildschirmdarstellung 108
Bildschirmmasken 133
Bildschirmspeicher 144, 289, 303, 306, 310
Bildschirmzeilen 22
BIOS-Routinen 289
Blockanweisung 26
Blockkommando 151
Blockstrukturen 23, 25
Boolesche Variable 67, 217, 279
BRUN20.EXE 16, 268 f., 272
BRUN20.LIB 16, 21, 268, 273, 275
BubbleSort-Routine 245

C
CALL 34, 39, 51, 55
CALL-Anweisung 42, 43, 45
CALL-Argumentliste 43 ff.
CALL-Argumentliste 47
CALL-Aufruf 50
CALL-Wert 45
CALL-Anweisung 52
CASE-Anweisung 286
CGA-Karte 303
CHDIR (change directory) 59
COLOR-Anweisung 56
COMMON-Anweisungen 56 ff.
Compiler 267
Cursor 122 f., 129 ff., 139
Cursorposition 111, 113, 123, 131

Cursorsteuerung 133
Cursortasten 117, 139, 177, 179

D
Dateinamen 14
Datenbereich 37, 41
Datenfeld 15, 51
Datenfeldelement 297
Datenfeldvariable 14
Datensegment 183 f., 297, 299 f.
Datenstrukturen 218
Datentyp 28
DEF FN 27
DEF-Typ 57
Deklarationsdatei 96 ff., 141
Demoprogramme 13
Descriptoren 300
Destination-Index 305
DIM 57
Dimension 59
Dimensionierung 223
Directory 263
Direktanwahl 146, 175, 198
Doppelpunkt 22
DOS-Funktionen 304
DOS-Interrupt 292

E
Editiermöglichkeiten 108, 110, 142
Editiertasten 114
Editor 18, 111
Einfügemodus 108, 114, 130, 142
Eingabemaske 108, 133
Eingabemodul 128
Eingabemodus 129
Eingabeposition 115
Eingaberoutine 32, 84, 98, 107 f., 124, 127 f.
Eingabeschleife 111, 113 f.
Eingabestring 111, 124
Eingabezeile 216
Eingabezone 107 f., 113 f., 120, 124, 129, 131, 214
Einzelmenüs 175
ELSEIF 24 ff.
END DEF 28
- IF 26
End of Subprogramm 33
END SUB 33
Endezeichen 129
Ereigniserfassung 20, 100
EXIT DEF 31
Extragsegment 305

F
Fallunterscheidung 24, 26
Farbadapter 303 ff.
Farbattribute 69
Farbeinstellung 56
Farben 116
Farbkarte 310
Farbvariablen 56, 58, 95
Farbwerte 55
Fehlerbehandlung 206, 224 f.
Fehlerbehandlungsroutine 205
Fehlernummer 225
Feld-Darstellungsmodus 134
Feldinhalt 108, 134
Feldlängen 134
Feldpositionen 134
Feldvorgaben 136 f.
Flag 67, 78, 80, 108, 134
Flagzustand 117
FOR..NEXT 23, 28
Funktionen 27
Funktionsargument 53
Funktionsaufruf 27
Funktionsbeschreibung 32
Funktionsdefinition 27 f.
Funktionskopf 30

G
Geschwindigkeit 100, 104
Globale Variable 55 f., 92 f.
GOSUB 18
GOTO 18
Grafikzeichen 165
Grenzfunktion 53
Groß-Klein-Schrift 208
Großbuchstaben 72

H
Hauptprogramm-Variable 35, 39 ff., 50
Hauptprogrammschleife 35
Hauptspeicher 88
Hauptspeicherkapazität 223
Herculeskarte 303
Hilfstexte 202
Hilfsvariable 157
Hintergrundfarbe 55, 113

I
IF-Blockanweisung 24
IF-Strukturen 13
IF..THEN..ELSE-Struktur 23
Initialisierungs-Datei 197
Initialisierungsteil 111
INKEY$ 78
INSTR-Funktion 30, 75
Integerarray 297, 312
Integervariable 65, 184, 300

K
Kleinbuchstaben 72
Kommandobreite 158
Kommandoende 157
Kommandonummer 162, 181
Kommandostring 181

Kommentaranzahl 215
Kommentarbefehle 249
Kommentare 18
Kommentarzeichen 208
Kompiliermenü 19
Kompilieroption 13, 19 f., 99, 205, 225, 270
Kompilierverfahren 99 f.
Kompiliervorgang 104
Konstanten 27, 49, 130, 145, 279, 286
Konstantenliste 286
Konvertierung 205
Konvertierungsprogramm 253

L

Label 13, 17 f., 207 f., 218, 220, 227, 233, 236, 238
Label-Suchfunktion 245
Labelanzahl 223
Labelende 248
Labelnummer 236, 241, 247
Labelsuche 247
Labeltabelle 237, 247 f.
Labelzeile 245
Laufzeit-Fehlermeldung 20
Laufzeitmodul 21, 268 f., 271, 275
Lesbarkeit 18
Lesezugriff 37
Library 15, 69, 83, 89, 266, 268, 272
Library-Batch 128
Linken 266, 269, 274 ff.
Linker 206, 272, 290
- aufrufen 20
Linkprogramm 266 f.
Linkvorgang 270
Logische Zeile 22
Lokale Variablen 29, 38, 41

M

Maschinenbefehle 266
Maschinencode 290
Maschinensprache-Befehle 37
Maschinensprache-Programm 19, 37, 38, 84, 85, 266
Maske 136
Maskeneditierung 138
Maskeneingabe 130, 134
Maskeninhalt 133
Maskensteuerung 98, 130, 133, 140 f.
Menü 148
Menü-Definition 151
Menüanwahl 179
Menüanzahl 176
Menüarray 175
Menübefehl 177
Menüinhalt 144
Menükommando 151
Menüleiste 153, 197
Menüname 148, 151 f., 159 f. 171, 197
Menünummer 171, 178
Menüparameter 164
Menürahmen 168
Menüstring 151, 153, 156 f., 161
Menüuntergrund 152, 182
MERGE-Anweisung 64
Metabefehl 57, 63, 228
MID$-Funktion 123
Modul 42, 83 f., 95, 108
Monochromadapter 303 ff.
Monochromkarten 310

O

Objektdatei 84, 89, 101, 104, 106, 144, 195, 266 f., 269, 272, 290
Objektmodul 99, 127, 141, 274, 290
Offset 292, 297 ff., 312
Offsetadresse 313
On Error 20
Optimierungsoptionen 100
OPTION BASE 57
Optionen 19
Optionsliste 99
Original-Bildschirminhalt 173
Original-Untergrund 181
Originalbildschirm 202
Originalvariable 51 f.

P

Parameter 49, 100, 103, 108, 134, 297
Parameterübergabe 49, 59, 291, 296
PATH-Anweisung 106
Pfadangabe 263
Pointer 287, 300
Positionszähler 115 f., 120
PRINT-Anweisung 61
PRINT-Befehl 145
Programm-Editor 267
Programm-Modul 94
Programmablaufplan 148
Programmänderung 19
Programmbibliothek 32
Programmdateien 102
Programmdiskette 101, 128, 195, 277
Programmiersprachen 35
Programmodule 205
Programmverfolgung 20
Programmzeile 21
Prozedur 15, 19, 33, 38, 46, 50, 95, 107, 290
Prozedur-Variable 38 f., 41, 47
Prozedurkopf 159
Prozedurname 14, 92
Prozessor 38
Prüfen zwischen Anweisungen 20
Pull-down-Menü 19, 84, 143, 145, 149, 155, 175, 189, 197, 205, 212, 216, 218, 224

Q

QB.Exe 16
Quelltext 19, 37

R
Rahmenbegrenzung 169
Rahmenbreite 165
Rahmenstring 165, 167
Rahmentyp 166, 168
Rahmenzeichen 158
Rahmenzeile 165
Records 287
Referenz 47
Register 311
Rekursion 287
REM-Anweisung 57, 64
Resultat 32
Resume Next 20
Richtungsflag 314
Routinen-Sammlung 95 f., 98, 113, 124, 129, 156, 177, 251
Rücksprungadresse 293, 298, 300

S
Schleifen 282
Schleifenanfang 280, 283
Schleifenanweisungen 123
Schleifenbildung 279
Schleifendurchgang 166
Schleifenende 280, 282
Schleifenstrukturen 280
Schleifenvariable 35, 66, 166
Schleifenzähler 236
Schlüsselwörter 23
Schnittstellen 291
Schreibanweisung 37
Schreibkonventionen 14
Schreibmodus 217
Schreibzugriff 37
Segment 292, 298, 311
SELECT CASE 285 f.
SHARED 57 f.
Sondertasten 67, 78, 116
Sonderzeichen 150
Sourcecode 144
Sourcetext 223
Speicherbereich 38, 183, 290
Speicherplatz 295
Speicherzelle 36 f., 46 f.
Sprunganweisungen 232
Sprungbefehl 218, 220, 232
Sprünge 220
Sprungziele 18
Stack 33, 44, 292, 295, 302
Stack-Speicher 38
Stackbehandlung 292
Stackbereich 311
Stackpointer 295 f., 301, 307
Stackverwaltung 293
Stammverzeichnis 102
Stand-alone-Porgramm 206, 268, 273, 290
Standard-Hintergrundfarbe 116
Standard-Library 89
Standard-Variablen 95, 98
Standard-Zeichenfarbe 116
Standardeingabe-Window 214, 216
Standardlabel 208, 238 f., 245
Standardoptionen 20, 100
Standardstrings 69
Stapel-Bereich 39
Stapel-Dateien 101
Stapel-Speicher 38, 41
Start-Position 107
Startadresse 301
Startposition 131
STATIC 34, 59 ff.
Steuerungsprozedur 143
String 15, 30, 75 f.
Stringadresse 299 f.
Stringanfang 122
Stringarray 149, 184, 219
Stringdescriptoren 299 f.
Stringende 120
Stringfunktion 27
Stringlänge 71, 299, 301
Stringvariable 14, 28
SUB-Anweisung 33, 42, 51 f., 55
SUB-Argumentliste 45, 47
SUB-Wert 45
SUB..END SUB 28
Subdirectory 101
Subprogramm 33
Suchschleife 77
Systemdateien 16, 265, 274
Systemeinrichtung 101, 277

T
Tabelle 36, 38, 220, 236, 268
Tastaturabfrage 78
Tastenvariablen 95, 124
THEN 26
Trennlinien 168
Trennstriche 150
Trennstring 149
Trennzeichen 120, 123
TROFF 20
TRON 20
Typdeklarationen 66

U
Übergabevariable 49
Übernahme-Array 51
Überschreibmodus 108, 114 ff., 130
Umlaute 73
Untergrund 177
Unterprogramm 15, 33
Unterprogramm-Deklaration 60
Unterroutine 15, 33
Unterstrich 22
User-Library 15, 19, 32, 83 ff., 90, 92 ff., 107

V
Variablen 27, 36, 49 f., 65, 68, 94 f., 107, 111, 130
Variablen-Deklaration 206
Variablen-Initialisierung 137
Variablenanzahl 188
Variablenart 29
Variablenliste 50
Variablenname 14, 18, 37, 46, 50, 251
Variablenorganisation 36
Variablentyp 45, 66
Variablenübergabe 42, 47, 49, 80
Variablenverwaltung 38
Verwaltungsmodule 152
Verzeichnisstruktur 102
Verzeichniswechsel 252
Videoadapter 303

W
Werte 55
Wertübergabe 43
Wertzuweisung 28
WHILE-Schleife 138, 157
WHILE..WEND 23, 28
Window-Breite 188, 211, 216
Window-Länge 188, 211, 216
Window-Routinen 199
Window-Verwaltung 199
Windowanzahl 186
Windowing 143, 310
Windowing-Routinen 205
Windows 183 ff., 210, 314
Windowzeile 314
Wortsprung 119
Wortzwischenraum 119

Z
Zeichenausgabe 289
Zeichenfarbe 55, 113
Zeichenfolge 15
Zeichenfolgevariable 14
Zeichenkette 28, 30
Zeichenposition 77
Zeichensatz 108
Zeilenmarke 18
Zeilennummern 17 f., 207 f., 219, 227, 232 f.
Zielzeilen 220
Zugriffspfad 59, 263
Zusatzkommentare 217
Zwei-Zeichen-Code 78

Für alle, die auf Nummer Sicher gehen wollen:

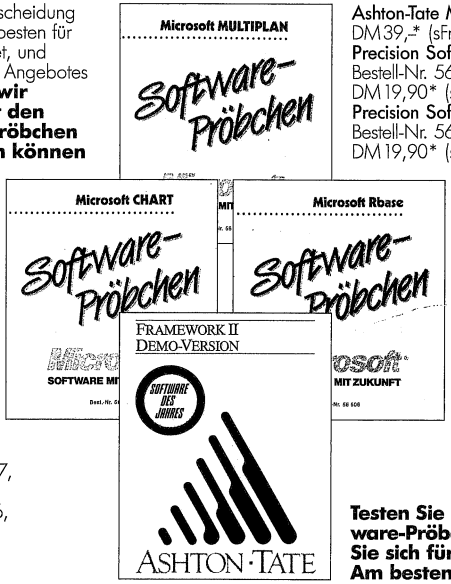
probieren Sie mal!

Software-Pröbchen von Word, Framework, dBASE und vielen anderen Programmen – jetzt bei Markt&Technik

Wenn Sie vor der schwierigen Entscheidung stehen, welche Software sich am besten für Ihre spezielle Problemlösung eignet, und Sie sich gerade über die Fülle des Angebotes informiert haben, **dann haben wir einen Super-Tip für Sie: Mit den Markt&Technik-Software-Pröbchen von aktuellen Programmen können Sie zu einem Preis ab nur DM 19,90* prüfen, was Ihren Anforderungen entspricht!**

Markt&Technik-Software-Pröbchen gibt es zu den Produkten:

Microsoft Word 3.0, Bestell-Nr. 56503, DM 39,-* (sFr 35,-*/öS 390,-*)
Microsoft Multiplan, Bestell-Nr. 56504, DM 39,-* (sFr 35,-*/öS 390,-*)
Microsoft Chart, Bestell-Nr. 56505, DM 39,-* (sFr 35,-*/öS 390,-*)
Microsoft Project, Bestell-Nr. 56507, DM 39,-* (sFr 35,-*/öS 390,-*)
Microsoft Rbase, Bestell-Nr. 56506, DM 39,-* (sFr 35,-*/öS 390,-*)
Ashton-Tate Framework, Bestell-Nr. 56502, DM 39,-* (sFr 35,-*/öS 390,-*)
Ashton-Tate dBASE III PLUS, Bestell-Nr. 56508, DM 39,-* (sFr 35,-*/öS 390,-*)



Ashton-Tate Multimate, Bestell-Nr. 56509, DM 39,-* (sFr 35,-*/öS 390,-*)
Precision Software Superbase Amiga, Bestell-Nr. 56517, DM 19,90* (sFr 18,90*/öS 199,-*)
Precision Software Superbase Schneider, Bestell-Nr. 56513, DM 19,90* (sFr 18,90*/öS 199,-*)
Precision Software Superbase Atari, Bestell-Nr. 56518, DM 19,90* (sFr 18,90*/öS 199,-*)
Pictures by PC, Bestell-Nr. 56515, DM 114,-* (sFr 65,-*/öS 1140,-*)
Timeline, Bestell-Nr. 56516, DM 114,-* (sFr 65,-*/öS 1140,-*)
Dataease, Bestell-Nr. 56511, DM 114,-* (sFr 65,-*/öS 1140,-*)
Enable, Bestell-Nr. 56514, DM 114,-* (sFr 65,-*/öS 1140,-*)

*Unverbindliche Preisempfehlung.

Testen Sie mit Markt&Technik-Software-Pröbchen – und entscheiden Sie sich für das richtige Programm. Am besten gleich bestellen!

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

709240

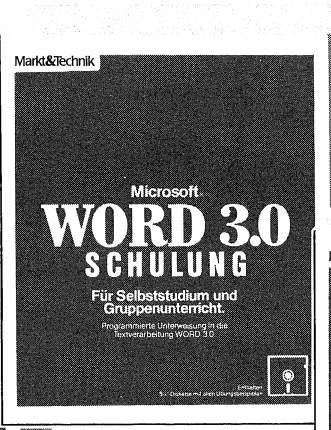
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

SCHULUNG

Praxiserprobte Arbeitsbücher zum Selbststudium und Gruppenunterricht für dBASE III PLUS, WORD 3.0 und LOTUS 1-2-3

- Alle theoretischen Grundlagen werden Ihnen mit praxisorientierten Beispielen und Übungen ausführlich erläutert.
- Dem Lehrmaterial liegt eine Diskette bei, die alle Übungsdateien in verschiedenen Bearbeitungsphasen enthält. So haben Sie jederzeit den sofortigen Anschluß an das Lehrprogramm.
- Für Ihre handschriftliche Beantwortung der Fragen ist ausreichend Platz vorgesehen. Später können Sie den »Frage-Teil« durch die Lösungen im Anhang ersetzen.
- Wenn Sie das Schulungspaket durchgearbeitet haben, bleibt Ihnen ein komplettes Nachschlagewerk einschließlich umfangreichem Befehlsverzeichnis für die tägliche Praxis.
- Software-Anforderung: dBASE III, WORD 3.0, bzw. LOTUS 1-2-3 Version 2

Dr. P. Albrecht
dBASE-III-PLUS-Schulung
1987, 547 Seiten,
inkl. Beispieldiskette
Best.-Nr. 90449
ISBN 3-89090-449-1
DM 98,–/sFr 90,20/öS 764,40



H. Niemeier
WORD-3.0-Schulung
1987, 456 Seiten,
inkl. Beispieldiskette
Best.-Nr. 90450
ISBN 3-89090-450-5
DM 98,–
sFr 90,20/öS 764,40

J. Steiner
LOTUS-1-2-3-Schulung Version 2
1987, 562 Seiten
inkl. Beispieldiskette
Best.-Nr. 90451
ISBN 3-89090-451-3
DM 98,–/sFr 90,20/öS 764,40

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser

706359

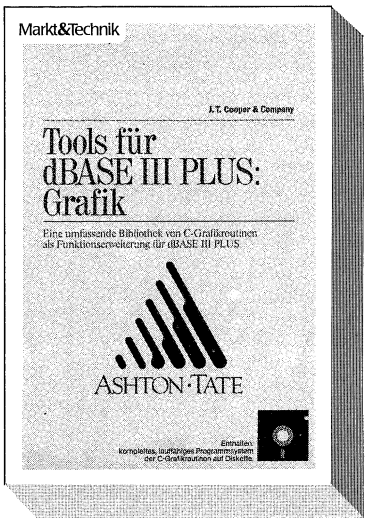
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

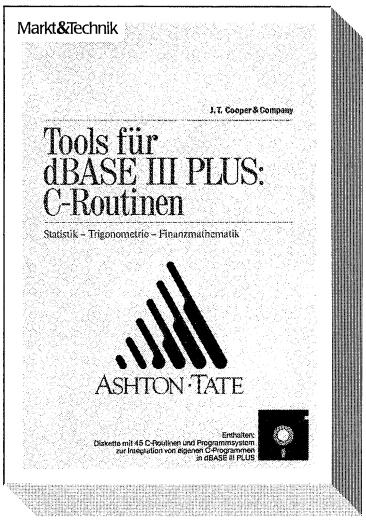
Ashton-Tate Edition bei Markt&Technik

Tools für dBASE III PLUS

Erweitern Sie den Funktionsumfang von dBASE III PLUS!

Tools – Utilities – Funktionen





Ashton-Tate
Tools für dBase III PLUS: Grafik
1987, 90 Seiten, inkl. Diskette
Mit dem Grafik-Band der Tools-Reihe können Sie auch anspruchsvolle Grafikfunktionen in die Programmiersprache von dBASE III PLUS integrieren, wie z.B. Torten-, Balken und XY-Diagramme.
Hardware-Anforderungen: IBM-PC/XT/AT oder hundertprozentig kompatibler Computer, 512 Kbyte RAM-Speicher mit mindestens zwei Diskettenlaufwerken.
Software-Anforderung: dBASE III PLUS
Best.-Nr. 90445, ISBN 3-89090-445-9
DM 98,- (sFr 90,20/öS 764,40)

Ashton-Tate
Tools für dBASE III PLUS: C-Routinen
1987, 97 Seiten, inkl. Diskette
Über 45 finanzmathematische, statistische und trigonometrische Funktionen sowie Kommandos zur Behandlung von Matrizen. Für den dBASE-Programmentwickler und C-Programmierer.
Hardware-Anforderungen: IBM-PC/XT/AT oder hundertprozentig kompatibler Computer, 512 Kbyte RAM-Speicher mit mindestens zwei Diskettenlaufwerken. Software-Anforderungen: Zum Modifizieren der Routinen und Hinzufügen neuer Funktionen ist ein C-Compilersystem erforderlich.
Best.-Nr. 90446, ISBN 3-89090-446-7
DM 98,- (sFr 90,20/öS 764,40)

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser

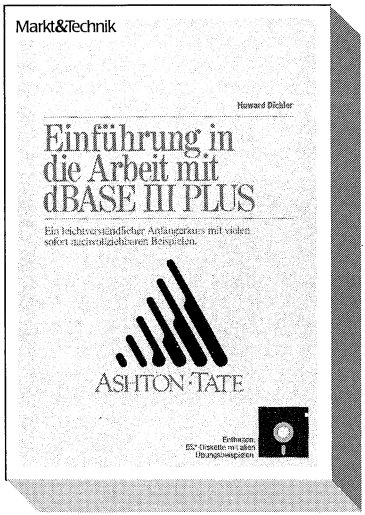
705361

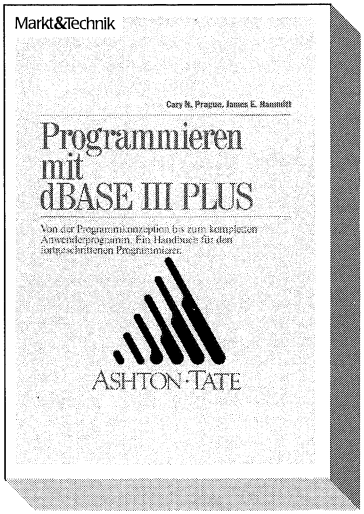
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Ashton-Tate Edition bei Markt&Technik

Profi-Literatur zu dBASE III

dBASE richtig und sinnvoll einsetzen!
Informationen – Techniken – Konzepte





Howard Dickler
Einführung in die Arbeit mit dBASE III PLUS
1987, ca. 500 Seiten, inkl. Diskette
Neben der Schilderung der verschiedenen Installationsroutinen für Disketten- und Festplattensysteme wird im ersten Teil vor allem Wert auf eine gründliche Einarbeitung des Lesers am Computer gelegt. Planen und Anlegen einer Datenbank stehen im Mittelpunkt des zweiten Teils. In einem weiteren Abschnitt geht es schließlich um die Programmierung mit dBASE III PLUS. Eine mitgelieferte Diskette enthält die wichtigsten Beispiele.
Hardware-Anforderungen: IBM-PC/XT/AT oder Kompatibler mit mindestens 512 Kbyte Arbeitsspeicher. Software-Anforderungen: dBASE III PLUS
Best.-Nr. 90468, ISBN 3-89090-468-8
DM 69,– (sFr 63,50/öS 538,20)

Cary N. Prague und James E. Hammitt
Programmieren mit dBASE III PLUS
1987, 376 Seiten
Ein unentbehrliches Nachschlagewerk für alle, die dBASE III PLUS professionell einsetzen wollen. Von der Benutzung des neuen ASSIST-Menüs bis zur Erstellung eines Programmsystems reicht das Spektrum dieses Buches. Schwerpunkte sind: die neuen Funktionen von dBASE III PLUS, Datenorganisation, Menüsysteme, Maskengestaltung, Programmoptimierung, Arbeit mit dem Programmgenerator.
Hardware-Anforderung: IBM-PC/XT/AT oder kompatibler Computer mit mindestens 384 Kbyte RAM. Software-Anforderung: dBASE III PLUS
Best.-Nr. 90469, ISBN 3-89090-469-6
DM 79,– (sFr 72,70/öS 616,20)

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computer-Fachgeschäften oder bei Ihrem Buchhändler.

707361

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Microsoft®

Junior-

Word. Das Kraftpaket zum Junior-Preis:

399,-*

Eine Idee macht Furore:
Word-Junior – das leistungsfähige, vielseitige und komfortable Textverarbeitungsprogramm für IBM-PCs und Kompatible. Zu einem unglaublichen Preis.

- Mit Word-Junior erledigen Sie mühelos Ihre gesamte Schreibarbeit im Büro oder im privaten Bereich.
- Word-Junior gibt Ihren Texten Format. Sie haben die Wahl zwischen verschiedenen Schrifttypen und -größen (je nach Drucker) sowie Absatz- und Druckformaten (Einrücken, Blocksatz, Flattersatz usw.).
- Gestalten Sie die Seiten ganz nach Ihrem persönlichen Geschmack (Seitenhöhe, Randbegrenzungen, Spaltenzahl, Zwischenräume usw.).
- Mit Word-Junior können Sie bis zu acht Dokumente parallel bearbeiten und gleichzeitig auf dem Bildschirm anzeigen. Wenn Sie Textteile zwischen diesen Dokumenten kopieren oder verschieben wollen, ist das für Word-Junior ein leichtes Spiel.
- Und mit der Maus werden Ihre Befehle noch schneller und in einem Arbeitsgang erledigt.

Überzeugen Sie sich selbst: Mit Word-Junior besitzen Sie eine Textverarbeitung der Luxusklasse!

Hardware-Anforderung:
IBM-PC oder kompatibler Computer, mindestens 192 Kbyte Speicher, zwei Diskettenlaufwerke oder Festplatte, MS-/PC-DOS ab Version 2.0.
Best.-Nr. 55111
DM 399,-*/sFr 345,-/öS 3990,-*

Ergänzende Literatur:
R. Wendel
Textverarbeitung mit Microsoft Word-Junior
(mit zusätzlichen Druckertreibern auf Diskette)
Best.-Nr. 90235
DM 49,-/sFr 45,10/öS 382,20*

Übrigens: In der Junior-Serie von Markt&Technik gibt es noch dBASE II, Framework, WordStar, Multiplan – ideale Standardsoftware für Einsteiger zum Junior-Preis.
Fragen Sie Ihren Computerfachhändler. Oder fordern Sie ausführliche Unterlagen direkt beim Verlag an.

*Unverbindliche Preisempfehlung.

Zeitschriften · Bücher
Software · Schulung

707264

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bestellungen im Ausland bitte an: SCHWEIZ: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Telefon (042) 415656
ÖSTERREICH: Rudolf Lechner & Sohn, Heizwerkstraße 10, A-1232 Wien, Telefon (0222) 677526
Ueberreuter Media Verlagsges. mbH (Großhandel), Laudongasse 29, A-1082 Wien, Telefon (0222) 481543-0.

ASHTON·TATE

Junior

Framework. Das Kraftpaket zum Junior-Preis:

399,-*

Dieses integrierte Programmpaket für PCs und kompatible Computer erfüllt alle Ansprüche, die an einen modernen »elektronischen Schreibtisch« gestellt werden:

- Textverarbeitung.

Ob Sie einen einseitigen Brief oder einen 50seitigen Bericht erstellen müssen: Mit wenigen Tastenanschlägen können Sie kopieren, löschen, umstellen.

- Tabellenkalkulation.

Wenn Sie wie die meisten Anwender nicht nur mit Worten, sondern auch mit Zahlen arbeiten, wissen Sie die starke Tabellenkalkulation in Junior-Framework zu schätzen.

- Datenbanksystem.

Beim Marktführer Ashton-Tate kann man vom Datenbankteil nur das Beste erwarten. Sie sortieren, aktualisieren, listen nach Kriterien Ihrer Wahl.

- Grafik.

Torten-, Balken- und Liniengrafik gehören zum Standardrepertoire von Junior-Framework.

- Konzepte.

Ein Ordnungsverfahren, mit dem Sie komplexe Aufgaben schnell und übersichtlich lösen.

- Programmiersprache FRED.

Mit dieser eingebauten Sprache erstellen Sie selbst komplexe Anwendungen für Ihren Bedarf. Dabei können Sie fast alle Funktionen der Teilbereiche von Junior-Framework verwenden.

Hardware-Anforderung:
IBM-PC oder kompatibler Computer, 384 Kbyte Speicher, zwei Diskettenlaufwerke oder ein doppelseitiges Diskettenlaufwerk und eine Festplatte, PC-/MS-DOS ab Version 2.0.
Best.-Nr. 55114
DM 399,-*/sFr 345,-/öS 3990,-*

Ergänzende Literatur:
R. Kost
Junior-Framework
Best.-Nr. 90500
DM 59,-/sFr 54,30/öS 460,20

Übrigens: In der Junior-Serie von Markt&Technik gibt es außerdem WordStar, dBASE II, Multiplan und Word – alle Produkte zum Junior-Preis.
Fragen Sie Ihren Computerfachhändler. Oder fordern Sie ausführliche Unterlagen direkt beim Verlag an.
*Unverbindliche Preisempfehlung.

707265

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0
Bestellungen im Ausland bitte an: SCHWEIZ: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Telefon (042) 41 56 56
ÖSTERREICH: Rudolf Lechner & Sohn, Heizwerkstraße 10, A-1232 Wien, Telefon (0222) 677526
Ueberreuter Media Verlagsges. mbH (Großhandel), Laudongasse 29, A-1082 Wien, Telefon (0222) 481543-0.

Computerliteratur und Software vom Spezialisten

Vom Einsteigerbuch für den Heim- oder Personalcomputer-Neuling über professionelle Programmierhandbücher bis hin zum Elektronikbuch bieten wir Ihnen interessante und topaktuelle Titel für

• Apple-Computer • Atari-Computer • Commodore 64/128/16/116/Plus 4 • Schneider-Computer • IBM-PC, XT und Kompatible
sowie zu den Fachbereichen Programmiersprachen • Betriebssysteme (CP/M, MS-DOS, Unix, Z80) • Textverarbeitung • Datenbanksysteme • Tabellenkalkulation • Integrierte Software • Mikroprozessoren • Schulungen.
Außerdem finden Sie professionelle Spitzen-Programme in unserem preiswerten Software-Angebot für Amiga, Atari ST, Commodore 128, 128 D, 64, 16, für Schneider-Computer und für IBM-PCs und Kompatible!
Fordern Sie mit dem nebenstehenden Coupon unser neuestes Gesamtverzeichnis und unsere Programmservice-Übersichten an, mit hilfreichen Utilities, professionellen Anwendungen oder packenden Computerspielen!

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

709005

Bitte schneiden Sie diesen Coupon aus, und schicken Sie ihn in einem Kuvert an:
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar

Adresse:

Name

Straße

Ort

Bitte schicken Sie mir:
☐ Ihr neuestes Gesamtverzeichnis
☐ Eine Übersicht Ihres Programmservice-Angebotes aus der Zeitschrift

☐ Außerdem interessiere ich mich für folgende/n Computer:

(PS: Wir speichern Ihre Daten und verpflichten uns zur Einhaltung des Bundesdatenschutzgesetzes)

Markt&Technik Verlag AG
– Unternehmensbereich Buchverlag –
Hans-Pinsel-Straße 2
D-8013 Haar bei München

Die ideale Ergänzung zur Fakturierung:

Finanzbuchhaltung

für IBM-PCs und Kompatible

Eine praxisnahe, menügesteuerte Sachkontenbuchhaltung für Klein- und Mittelbetriebe.
Per Tastendruck rufen Sie die aktuelle Bilanz oder den Kontostand ab und sind dadurch jederzeit über Ihre Finanzlage informiert. Ein schnelles, bedienerfreundliches Buchen ist durch die Belegung der Funktionstasten garantiert. Features wie Kostenstellenauswertung, Gewinn+Verlust-Rechnung, Bilanzierung und automatische Umsatzsteuervoranmeldung machen dieses Programm zu einem leistungsstarken Partner im Büro. Die Fakturierung für IBM-PCs und Kompatible, Verlag Markt&Technik, enthält eine Offene-Posten-Verwaltung. Die Salden können Sie in die Finanzbuchhaltung übernehmen. Bitte beachten Sie, daß bei diesem Programm buchhalterische Kenntnisse vorausgesetzt werden.

Hardware-Anforderungen:
IBM-PC oder kompatibler Computer unter PC-DOS ab Version 2.0 bzw. MS-DOS ab Version 2.11, zwei Diskettenlaufwerke oder ein Diskettenlaufwerk mit zusätzlicher Festplatte mindestens 256 Kbyte RAM-Speicher.

Bestell-Nr. 56106, 5¼-Zoll-Diskette

DM 249,–*

(sFr 229/öS 2490,–*)

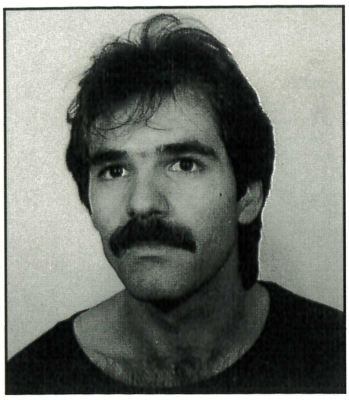
* inkl. MwSt. Unverbindliche Preisempfehlung

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

709239-2

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0



SAID BALOUI
Der Autor beschäftigt sich seit 1983 mit Heim- und Personalcomputern. Aus seiner Feder stammen Bücher, Artikel und Programme, die sich mit der Programmierung dieser Computer sowohl in BASIC als auch in Assembler befassen.

Effektives Programmieren mit QuickBASIC

Mit QuickBASIC wird die Programmiersprache BASIC endlich »erwachsen«. Dies betrifft nicht nur die Ausführungsgeschwindigkeit, sondern vor allem die höchst effizienten Möglichkeiten zur Strukturierung von Programmen: Prozeduren, Blockstrukturen, lokale und globale Variablen, User-Libraries und benutzerdefinierte Funktionen.
Der Umgang mit diesen neuen Eigenschaften erfordert jedoch eine beträchtliche Umstellung von Ihnen. Und dabei wird Ihnen dieses Buch helfen, das alle neuen »Features« problemorientiert behandelt. Zu Beginn werden die grundlegenden Eigenschaften von QuickBASIC anhand kleiner Demoprogramme erläutert. Anschließend folgen immer komplexer werdende Programm-Module: eine Eingaberoutine, ein Modul zur Verwaltung von Eingabemasken, ein weiteres zur Verwaltung von Windows und Pull-down-Menüs. Anhand dieser umfangreichen Beispiele erläutert Ihnen der Autor detailliert den Umgang mit statischen Variablen, dynamischen Arrays und vor allem den User-Libraries von QuickBASIC.
Den Höhepunkt des BASIC-Teils bildet das Programm CONVERT, ein Anwendungsprogramm, mit dem Sie Ihre GW-BASIC-Programme in Quick-BASIC-»Format« konvertieren können. Der letzte Teil des Buches ist besonders für die Assemblerfreaks unter Ihnen interessant. Er befaßt sich mit der Einbindung von Assembler-Programmen, dem Programmaufbau und der Variablenübergabe von/an QuickBASIC.
Alle entwickelten Programme sind auf der beiliegenden Diskette im Quell- und im Objektcode vorhanden.

Hardware-Anforderungen:

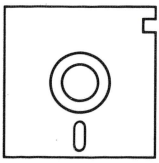
- Personalcomputer (PC/XT/AT) mit einem doppelseitigen Floppylaufwerk (360 Kbyte)
- Mindestens 256 Kbyte freier Arbeitsspeicher bei QuickBASIC Version 2.0
- Mindestens 320 Kbyte freier Arbeitsspeicher bei QuickBASIC Version 3.0

Software-Anforderungen:

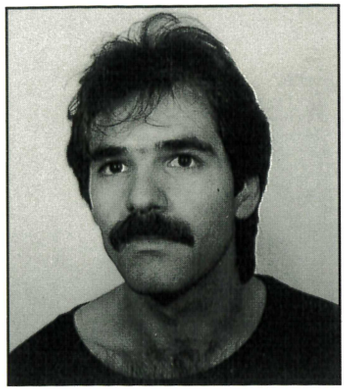
- QuickBASIC in Version 2.0 oder 3.0 von Microsoft

ISB N 3-89090-532-3

Markt&Technik



DM 69,-
sFr 63,50
öS 538,20



SAID BALOUI
Der Autor beschäftigt sich seit 1983 mit Heim- und Personalcomputern. Aus seiner Feder stammen Bücher, Artikel und Programme, die sich mit der Programmierung dieser Computer sowohl in BASIC als auch in Assembler befassen.

Effektives Programmieren mit QuickBASIC

Mit QuickBASIC wird die Programmiersprache BASIC endlich »erwachsen«. Dies betrifft nicht nur die Ausführungsgeschwindigkeit, sondern vor allem die höchst effizienten Möglichkeiten zur Strukturierung von Programmen: Prozeduren, Blockstrukturen, lokale und globale Variablen, User-Libraries und benutzerdefinierte Funktionen.
Der Umgang mit diesen neuen Eigenschaften erfordert jedoch eine beträchtliche Umstellung von Ihnen. Und dabei wird Ihnen dieses Buch helfen, das alle neuen »Features« problemorientiert behandelt. Zu Beginn werden die grundlegenden Eigenschaften von QuickBASIC anhand kleiner Demoprogramme erläutert. Anschließend folgen immer komplexer werdende Programm-Module: eine Eingaberoutine, ein Modul zur Verwaltung von Eingabemasken, ein weiteres zur Verwaltung von Windows und Pull-down-Menüs. Anhand dieser umfangreichen Beispiele erläutert Ihnen der Autor detailliert den Umgang mit statischen Variablen, dynamischen Arrays und vor allem den User-Libraries von QuickBASIC.
Den Höhepunkt des BASIC-Teils bildet das Programm CONVERT, ein Anwendungsprogramm, mit dem Sie Ihre GW-BASIC-Programme in Quick-BASIC-»Format« konvertieren können. Der letzte Teil des Buches ist besonders für die Assemblerfreaks unter Ihnen interessant. Er befaßt sich mit der Einbindung von Assembler-Programmen, dem Programmaufbau und der Variablenübergabe von/an QuickBASIC.
Alle entwickelten Programme sind auf der beiliegenden Diskette im Quell- und im Objektcode vorhanden.

Hardware-Anforderungen:
- Personalcomputer (PC/XT/AT) mit einem doppelseitigen Floppylaufwerk (360 Kbyte)
- Mindestens 256 Kbyte freier Arbeitsspeicher bei QuickBASIC Version 2.0
- Mindestens 320 Kbyte freier Arbeitsspeicher bei QuickBASIC Version 3.0

Software-Anforderungen:
- QuickBASIC in Version 2.0 oder 3.0 von Microsoft

ISB N 3-89090-532-3

Markt&Technik

4 001057 905320

DM 69,-
sFr 63,50
öS 538,20

Markt&Technik

Effektives Programmieren mit QuickBASIC

Baloui

90532

Markt&Technik

Said Baloui

Effektives Programmieren mit QuickBASIC

Eine systematische Anleitung zum Entwickeln von effizienten und professionellen Programmen unter Microsoft QuickBASIC.

Auf Diskette im PC-/MS-DOS-Format enthalten: alle Programm-Module aus dem Buch, wie z. B. Window-Verwaltung, Pull-down-Menüs, schnelle Bildschirmausgabe, Eingabemasken-Verwaltung und »CONVERT«, einem Programm, das Ihre GW-BASIC-Programme nach QuickBASIC konvertiert.

Markt&Technik

Zeitschriften · Bücher

Software · Schulung

Markt&Technik
Verlag Aktiengesellschaft
Hans-Pinsel-Straße 2
8013 Haar bei München

Diskette zum Buch
Effektives Programmieren mit QuickBASIC
IBM 360 KB Format
Bestell-Nr. 90532 / 16118746

Das auf diesem Datenträger enthaltene Programm ist urheberrechtlich geschützt. Unerlaubte Kopierung, Vervielfältigung, Verleih oder Vermietung ist untersagt. Jegliche unautorisierte Nutzung wird straf- und zivilrechtlich verfolgt.

Markt&Technik

Zeitschriften · Bücher

Software · Schulung